Publikationsreihe des John von Neumann-Instituts für Computing (NIC)
NIC Serie Band 28

John von Neumann-Institut für Computing (NIC)

Jörg Striegnitz

Integration von Programmiersprachen durch strukturelle Typanalyse und partielle Auswertung

NIC Serie Band 28

ISBN 3-00-016006-X

Die Deutsche Bibliothek – CIP-Einheitsaufnahme
Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

Herausgeber: NIC-Direktorium

Vertrieb: NIC-Sekretariat
Forschungszentrum Jülich
52425 Jülich
Deutschland

Internet: www.fz-juelich.de/nic

Druck: Graphische Betriebe, Forschungszentrum Jülich

NIC Serie Band 28 ISBN 3-00-016006-X

Integration von Programmiersprachen durch strukturelle Typanalyse und partielle Auswertung

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

von
Diplom-Informatiker Jörg Striegnitz
aus Thuine (Lingen/Ems)

Berichter: Universitätsprofessor Dr. rer. nat. Klaus Indermark
Universitätsprofessor Dr. rer. nat. Friedel Hoßfeld

Tag der mündlichen Prüfung: 21. Dezember 2004

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar

Meinem lieben Vater, Martin Striegnitz, der viel zu früh verstorben ist, meiner Mutter, Marianne Striegnitz, die mir zusammen mit meinem Vater eine wunderbare Kindheit und Jugend ermöglicht hat, sowie meiner lieben Frau Esther, und meinen beiden Söhnen Jan und Erik, die am meisten unter der Erstellung dieser Arbeit gelitten haben, und die mir stets Kraft zum Durchhalten gegeben haben.

Jörg Striegnitz, August 2004

Abstract

The choice of a programming language quite often depends on the problem domain. Examples are the use of object-oriented languages for distributed systems, the use of functional languages in compiler construction, or the use of logical programming languages in artificial intelligence projects. In the extreme case it even makes sense to develop a domain specific language.

In larger software projects it is desirable to implement each module in the programming language which is best suited for the specific module's task. Of course, this raises the question of how to integrate those modules to a coherent, working and efficient overall system.

This dissertation focuses on a special case of language integration: the embedding of a language in an existing one. A new embedding-technique is proposed, based on structural type analysis and partial evaluation.

In the first part of this thesis a set of three model-languages will be introduced. All these languages are designed to support our new embedding-approach, which will be thoroughly explained. The properties of the model-languages and the restrictions they impose on the guest languages will then be explained in full detail.

As a first result, it turns out that many concepts of the model-languages can be simulated in C++ and thus, these languages are well-suited to explain the phenomena of the C++ -Template-Metaprogramming technique.

The second part of this thesis analyses the practical relevance of our new embedding technique. We will show how to integrate a functional programming language with lazy evaluation, garbage collection and algebraic datatypes into C++ . We will show that our approach allows for the generation of code that is nearly as efficient as code being generated by established Haskell compilers.

Kurzfassung

Die Wahl einer Programmiersprache ist häufig vom zu lösenden Problem motiviert (z.B. objektorientierte Sprache für verteilte Systeme, funktionale Sprache für Übersetzer, logische Sprache im Bereich der künstlichen Intelligenz). Im Extremfall kann es sich sogar lohnen, für ein Softwareprojekt eine neue, problemspezifische Sprache (engl.: *domain specific language*) zu entwerfen.

Bei größeren Softwareprojekten kann es sich anbieten, einzelne Module in der Programmiersprache zu implementieren, die für den Einsatzzweck des Moduls am besten geeignet ist. Dies wirft natürlich sofort die Frage auf, wie man diese Module zu einem kohärenten, funktionierenden und effizienten Gesamtsystem verbindet.

Diese Dissertation beschäftigt sich mit einem Spezialfall der Sprachintegration: der Einbettung einer Gastsprache in eine Wirtssprache und schlägt ein neues Verfahren zur Spracheinbettung vor, welches auf struktureller Typanalyse und partieller Auswertung beruht.

Im ersten Teil dieser Arbeit werden drei Modell-Gastsprachen eingeführt, die unseren Ansatz zur Spracheinbettung unterstützen. Die Eigenschaften dieser Sprachen werden ausführlich erklärt und es wird untersucht, welche Restriktionen sie der Gastsprache auferlegen.

Es wird sich zeigen, daß sich viele Konzepte der Modell-Gastsprachen in C++ simulieren lassen und diese sich damit auch zur Erklärung der Phänomene der C++ -Template-Metaprogrammierung eignen.

Im zweiten Teil dieser Dissertation wird die Tragfähigkeit unseres Integrationsansatzes an einem praktischen Beispiel untersucht: der Integration einer funktionalen Sprache mit verzögerter Auswertung, garbage collection und algebraischen Datentypen in C++ .

Es wird sich herausstellen, daß unsere Einbettungstechnik das Erzeugen von effizientem Programmcode erlaubt, der sogar mit etablierten Haskell-Übersetzern mithalten kann. Am Beispiel der Optimierung von endrekursiven Aufrufen, einer Optimierung, die von den meisten C++ -Übersetzern nicht vorgenommen wird, wird klar, dass sich der Ansatz auch zur Implementierung von domänenspezifischen Optimierungen eignet.

Danksagung

Meinen beiden Gutachtern, Professor Dr. Klaus Indermark und Professor Dr. Friedel Hoßfeld, danke ich dafür, daß sie mir diese Doktorarbeit ermöglicht haben, daß sie mir viel Freiraum zu ihrer Entwicklung gegeben haben und mich stets mit Rat und Tat unterstützt haben.

Diese Doktorarbeit entstand am Zentralinstitut für Angewandte Mathematik (ZAM) im Forschungszentrum Jülich. Herrn Professor Dr. Hoßfeld, der die überwiegende Zeit meiner Tätigkeit Direktor des ZAM war, und seinem Nachfolger, Herrn Professor Dr. Dr. Lippert, danke ich für die hervorragenden Arbeitsbedingungen, die ich am ZAM vorfinden durfte.

Bei allen Mitarbeiterinnen und Mitarbeitern des ZAM möchte ich mich dafür bedanken, daß sie mir jederzeit bereitwillig geholfen haben, mich unterstützt haben, und dafür, daß ich auch in schwierigen Zeiten viel mit ihnen lachen durfte. Besonders herausheben möchte ich meinen Abteilungsleiter, Dr. Rüdiger Esser, der mir während meiner Arbeit an dieser Dissertation stets den Rücken freigehalten hat, Herrn Dr. Bernd Mohr, dem ich einen Großteil meines Wissens über die Programmiersprache C++ verdanke, und Herrn Dr. Bernhard Steffen, der für meine mathematischen „Problemchen" stets ein offenes Ohr hatte.

Herrn Professor Dr. Wolfgang Nagel danke ich dafür, daß ich zahlreiche Tutorien über C++ am Zentrum für Hochleistungsrechnen der TU Dresden abhalten konnte. Die vielen Fragen und Anregungen aus dem Auditorium haben einen großen Einfluss auf mein Verständnis von C++ gehabt und damit entscheidend zum Gelingen dieser Arbeit beigetragen.

Danken möchte ich auch vielen internationalen Kollegen, die mir in zahlreichen Diskussionen Anregungen und Hinweise gaben, die in diese Arbeit eingeflossen sind. Danke an: Kim Bruce, Timothy Budd, Charles Consel, Krzysztof Czarnecki, Kei Davis, Erik Ernst, Michael Gerndt, Kevin Hammond, Jakkoo Järvi, Herbert Kuchen, Sybille Schupp, Yannis Smaragdakis, Stephen Smith, Craig Rasmussen, Walid Taha, Todd Veldhuizen und alle, die ich vergessen habe.

Tatjana Eitrich schulde ich Dank für ihre Zuverlässigkeit und dafür, daß sie mir in mehreren Iterationen half, diese Arbeit orthographisch zu optimieren. Danke auch an meinen Studien- und Bürokollegen Karsten Scholtyssik für viele interessante Diskussionen während unserer gemeinsamen Zeit am ZAM.

Meiner größter Dank gilt meiner Frau und meinen Kindern, die immer für mich da waren, mich unterstützt und motiviert haben, und allzu oft Geduld zeigten, wenn die Arbeit mal wieder „fast fertig" war.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Nahezu jeder Softwareentwickler, der mehrere Programmiersprachen und -methoden beherrscht, wird sicherlich schon einmal in die Situation gekommen sein, in der er ein Projekt in Programmiersprache *A* begonnen hat, und an einem gewissen Punkt im Projektverlauf feststellt, daß sich ein Teilproblem viel besser mit den Konzepten aus Programmiersprache *B* lösen ließe.

Hat man die freie Wahl, so würde man sich zur Realisierung eines Parsers wahrscheinlich für eine funktionale Sprache entscheiden, da algebraische Datentypen zusammen mit dem Mechanismus des *pattern matching* eine sehr übersichtliche und nachvollziehbare Implementierung erlauben. Design, Modell und Programmcode lassen sich so hervorragend aufeinander abbilden.

Geht es jedoch um die Programmierung einer graphischen Benutzeroberfläche (GUI - *Graphical User Interface*), so findet man bei objektorientierten Programmiersprachen geeignetere Konzepte. Die in hierarchischer Form organisierten Kontrollelemente einer GUI (z.B. *Button* mit und ohne *Icon*) lassen sich sehr intuitiv durch Klassenhierarchien modellieren, weshalb auch hier Design, Modell und Programmcode sehr gut korrespondieren.

Expertensysteme erfordern in der Regel sehr spezialisierte Suchverfahren und machen regen Gebrauch von Resolution und Backtracking. Eine logische Programmiersprache bringt diese Techniken von Haus aus mit und qualifiziert sich damit als Sprache der Wahl für solche Systeme.

Selbstverständlich kann man auch in funktionalen Sprachen eine GUI programmieren, jedoch muß man die ihr inhärenten objektorientierten Konzepte mühsam simulieren. Genauso kann man ein Expertensystem zwar auch in einer imperativen Programmiersprache realisieren, muß aber auch hier die Kernkonzepte von Hand programmieren. Die zu Grunde liegende Programmiermethodik läßt sich dann nur noch schwer aus dem Quelltext ablesen und macht ihn unübersichtlich und schwer lesbar. Da der Übersetzer kein Wissen über die simulierten Konzepte hat, kann er deren Mißbrauch nicht erkennen (fehlendes Typsystem) und auch keine spezifischen Optimierungen vornehmen.

Grobgranular betrachtet, lassen sich Programmiersprachen in die Klassen objektorientiert, funktional, logisch und imperativ einteilen. Man spricht auch von **Programmierparadigmen**. Ein Programmierparadigma gibt gewissermaßen ein *Denkmodell* vor [144], in dem der Programmierer seine Lösung entwickelt und nimmt damit Einfluß auf die frühen Phasen des

Softwareentwicklungsprozesses (Design- und Entwurfsphase; z.B. objektorientierte Analyse und Entwurf).

Problemangepaßte Sprachen (engl.: *domain specific languages* - DSLs) gehen noch einen Schritt weiter: Sie wurden speziell für den Einsatz in ganz bestimmten Problemfeldern (engl.: *domains*) entwickelt. Sie erheben selten den Anspruch berechnungsuniversell zu sein, bieten dafür aber eine auf das Aufgabengebiet zugeschnittene Syntax und Semantik. Darüber hinaus nehmen Übersetzer für DSLs domänenspezifische Optimierungen vor, die ein normaler Übersetzer mangels Wissen nicht durchführen könnte. Ein sehr prominentes Beispiel für eine DSL ist die *Structural Query Language* (SQL), die speziell für den Umgang mit Datenbanken entwickelt wurde.

Die extreme Spezialisierung dieser Sprachen verspricht einen kürzeren Weg vom Design zur Implementierung und bessere Eigenschaften hinsichtlich der Wiederverwendbarkeit von Programmcode. Darüber hinaus können DSL-Übersetzer in der Regel effizienteren Binärcode erzeugen, als solche für Universalsprachen (engl.: *general purpose languages*).

Im weiteren Sinne kann man auch Programmbibliotheken, etwa das *Message Passing Interface* MPI [58], oder die *Basic Linear Algebra Subroutines* (BLAS) [64] als DSLs auffassen. Allerdings wird hier nur bedingt eine feste Syntax vorgegeben. In der Regel werden ungültige Funktionskompositionen (z.B. die Multiplikation einer 2×4 Matrix mit einer 6×7 Matrix) erst zur Laufzeit erkannt und der Übersetzer nimmt keine domänenspezifischen Optimierungen vor, da ihm die Bedeutung der einzelnen Unterroutinen nicht bekannt ist.

Es sind jedoch nicht nur softwaretechnische Aspekte, die bei der Wahl einer Programmiersprache eine Rolle spielen. Im Umfeld des *High-Performance Scientific Computing* ist man insbesondere an der Effizienz des generierten Binärcodes interessiert. Abhängig von der Zielhardware weisen diverse Programmiersprachen unterschiedliche Qualitäten auf. Funktionale Sprachen, die frei von Seiteneffekten sind, eignen sich z.B. gut zur Programmierung von Datenfluß- oder Vektorrechnern (z.B. die funktionale Sprache Sisal [27]). Die spätestens seit Smalltalk etablierte Sicht von kommunizierenden Objekten[1] macht objektorientierte Sprachen zu interessanten Werkzeugen der verteilten Programmierung – es verwundert daher kaum, daß verteilte Programmierwerkzeuge wie Corba oder COM/DCOM auf einem objektorientierten Ansatz beruhen. Im Vergleich dazu lassen sich imperative Sprachen sicher am einfachsten in den Maschinencode einer von Neumann Maschine übersetzen, da ihr Ausführungsmodell von diesem Maschinenmodell abgeleitet wurde. Bereits in den frühen achtziger Jahren hat man erkannt, daß die erreichbare Ausführungsgeschwindigkeit auf einer Zielhardware abhängig von den Eigenschaften einer Programmiersprache ist (FORTRAN77-Code läßt sich z.B. einfacher vektorisieren [95, 96] als C-Code, da es in FORTRAN keine *aliasing*-Probleme gibt).

Ein größeres Softwareprojekt durchgängig in einer Programmiersprache zu realisieren scheint nicht immer die beste Wahl zu sein. Denken wir beispielsweise an eine integrierte Entwicklungsumgebung, die eine graphische Benutzeroberfläche, einen Übersetzer, ein lernfähiges Optimierungssystem und eine Codedatenbank enthält. Oder denken wir an ein heterogenes *Grid* [48], in dem sowohl Vektor-, als auch massiv-parallele Systeme, SMP-Cluster und rekonfigurierbare Einheiten vernetzt sind. Welche Programmiersprache ist für solche Projekte die beste?

Es scheint wünschenswert, Teilprobleme mit den optimalen Programmierwerkzeugen zu lösen, und auf die am besten geeignete Hardware abzubilden. Spätestens bei der Komposition des Gesamtsystems stellt sich dann die Frage, wie die einzelnen Module Daten austauschen und

[1] Der Aufruf einer Methode wird als das Senden einer Nachricht interpretiert.

wie sich der Kontrollfluß von Modul zu Modul bewegen kann. Darüber hinaus bringen nahezu alle Programmiersprachen spezielle Entwicklungswerkzeuge mit, so daß die Entwicklung eines Projektes unter der Verwendung mehrerer Sprachen zu einem echten Problem werden kann.

1.2 Kombination von Programmiersprachen

Zahlreiche Ansätze zur Komposition von Modulen, die in unterschiedlichen Programmiersprachen entwickelt wurden, und zur Integration von Programmierkonzepten sind bereits beschritten worden. Grob betrachtet lassen sie sich in folgende vier Klassen einteilen:

- **Spezialsprachen**, die Konzepte mehrerer Programmierparadigmen in sich vereinen; z.B.
  - Oz [120] (logisch / objektorientiert)
  - Curry [65] (funktional / logisch)
  - PIZZA [128] und Brew [14] (imperativ, objektorientiert und funktional)
  - Objective ML [145] (funktional / objektorientiert)
  - Skil [17] (imperativ und funktional)
  - Leda [24] und J/MP [25] (funktional, logisch, imperativ und objektorientiert).

  Dazu gehören aber auch **Domain Specific Languages**; z.B. SQL, Postscript, die Hardwarebeschreibungssprachen VHDL und VERILOG und natürlich die zum Satz dieser Arbeit verwendete Sprache LaTeX.

- **Multilinguale Systeme**, mit denen sich in verschiedenen Sprachen verfaßte Module koppeln lassen, z.B.:
  - UNIX-pipes [61]
  - Spezielle Koordinationssprachen [181]
  - Frameworks für das verteilte Rechnen [69] (z.B. Corba [16], DCOM [59] und MPI [58])
  - *Foreign Function Interfaces* (FFI) (z.B. Haskell GreenCard [138], H/Direct [45] das Prolog FFI [103], oder das JAVA Native Language Interface (JNI) [104]).
  - Übersetzung der Module in die Zielsprache einer virtuellen Maschine (z.B. UNCOL [153][37], Pascal-Code [121], JAVA Virtual Maschine (JVM [105] [44]), oder das `.NET` Framework von Microsoft [18])

- **Rahmenwerke** (engl.: *Frameworks*) zur Unterstützung von bestimmten Programmierstilen; z.B. die Rahmenwerke von Läufer [99] und Striegnitz [155] zur Integration von Funktionen höherer Ordnung und *Currying* in C++ , oder die von McNamara und Smaragdakis entwickelte Bibliothek FC++ zur Bereitstellung von Funktionen höherer Ordnung und Listen mit verzögerter Auswertung [111].

- **Eingebettete Programmiersprachen**. Hier wird eine neue Sprache (die eingebettete Sprache oder auch Gastsprache) in eine bestehende Sprache (die Hostsprache) eingebettet. Voraussetzung ist, daß die Gastsprache syntaktisch gesehen eine Teilmenge der

Hostsprache ist. Die Idee ist, Objekte der eingebetteten Sprache auf Funktionen der Hostsprache abzubilden, so daß Programme der eingebetteten Sprache sich als Komposition dieser Funktionen darstellen. Diese Funktionen führen Berechnungen allerdings nicht unmittelbar aus, sondern generieren einen abstrakten Syntaxbaum (engl.: *Abstract Syntax Tree* - AST), der später von einem Interpreter ausgewertet wird.

Beispiele finden sich im Umfeld von C++ ; etwa die Einbettung eines einfachen λ-Kalküls in C++ von Striegnitz [157], oder die Einbettung einer imperativen Sprache in C++ durch Järvi [79]. Besonders beliebt ist die Einbettung von DSLs in funktionale Sprachen. Beispiele finden sich in den Bereichen *Parsing* [110, 76, 130], *pretty printing* [75], Grafik [46, 43], Musik [72], Robotik [134], grafische Benutzeroberflächen [38], Datenbanken [100] und dem Hardware-Design [15, 109, 129].

Spezialsprachen haben den Nachteil, daß sie nicht erweiterbar sind und in der Regel nur auf wenigen Plattformen zur Verfügung stehen. Ihr Vorteil ist, daß sie speziell auf den Paradigmen-Mix bzw. das intendierte Problemfeld zugeschnitten sind und ihre Übersetzer spezifische Optimierungen bereitstellen können.

Multilinguale Systeme koordinieren in erster Linie den Datenaustausch zwischen mehreren Sprachen. Dieser ist meistens auf Grundtypen eingeschränkt, was zu erheblichem Programmieraufwand beim Austausch von komplexeren Datentypen führt. Besonders erschwert wird der Datenaustausch, wenn Daten untrennbar mit Programmcode verbunden sind (z.B. Objekte oder Funktionsabschlüsse), oder unterschiedliche Strategien zur Speicherverwaltung eingesetzt werden (etwa bei Sprachen mit und ohne *garbage collection*). Sprachübergreifende Optimierungen sind nicht möglich.

Rahmenwerke sind zwar hilfreich, wenn es darum geht, bestimmte Programmiertechniken in einer Sprache einfacher zugänglich zu machen, können eine Sprache mit nativer Unterstützung für diese Konzepte jedoch nicht ersetzen, da zum Beispiel ein spezialisiertes Typsystem fehlt und keine der Programmiertechnik angepaßten Optimierungen vollzogen werden.

Der Ansatz der Spracheinbettung bietet hier Vorteile: Dadurch, daß das eingebettete Programm als AST vorliegt, können semantische Tests und Optimierungen vorgenommen werden. Ferner erlaubt er die simultane Integration mehrerer Sprachen und ist erweiterbar.

Die Herausforderung liegt hier im Design der Hostsprache. Um Typfehler in einem Programm, welches in der Gastsprache verfaßt wurde, möglichst früh zu erkennen, muß man unzulässige Funktionskompositionen durch das Typsystem der Hostsprache ausschließen. Ansonsten könnten Typfehler im Gastprogramm erst zur Laufzeit, wenn der Typchecker der Gastsprache ausgeführt wird, erkannt werden – der AST für das Gastprogramm liegt ja erst zur Laufzeit vor.

1.3 Ziele dieser Arbeit

Im Rahmen dieser Arbeit wird ein neuer Ansatz zur Einbettung von Programmiersprachen vorgestellt. Dieser beruht auf der innovativen Kombination zweier Techniken in einer Programmiersprache: der **strukturellen Typanalyse** [67] und der **partiellen Auswertung** [51].

Grob skizziert stellt sich der Ansatz wie folgt dar: Die Gastsprache wird durch Überladung von Funktionen in die Hostsprache eingebettet. Dabei werden die Funktionen so überladen, daß

ein AST für das jeweilige Programmfragment der Gastsprache generiert wird. Das besondere an diesem AST ist, daß man seinen strukturellen Aufbau aus dem Namen seines Datentyps ablesen kann.

Durch spezielle Sprachkonstrukte (strukturelle Typanalyse) kann die im Typ sichtbare Struktur des AST analysiert werden, um darauf aufbauend eine semantische Analyse und eine Übersetzungsfunktion von der Gastsprache in die Hostsprache zu implementieren. Typabhängige Operationen werden durch partielle Auswertung zur Übersetzungszeit der Hostsprache abgearbeitet, so daß man den Übersetzer der Hostsprache effektiv zur Übersetzung der Gastsprache in Maschinensprache einsetzen kann.

Wir stellen folgende Hypothesen auf:

H1 Strukturelle Typanalyse, gepaart mit partieller Auswertung, ist ein geeignetes Werkzeug zur Einbettung einer getypten Sprache in eine Hostsprache.

H2 Diese neue Technik der Spracheinbettung erlaubt prinzipiell die Übersetzung der Gastsprache in laufzeiteffizienten Maschinencode.

H3 Unsere Methode ist geeignet, die Phänomene der C++ -Template-Metaprogrammierung zu beschreiben.

Zum Beweis dieser Thesen gehen wir wie folgt vor: Wir werden zunächst eine Modellsprache entwickeln, die das Konzept der strukturellen Typanalyse unterstützt und einen partiellen Auswerter für diese Sprache vorstellen, mit dem Berechnungen, die im Zusammenhang mit der strukturellen Typanalyse stehen, in die Übersetzungszeit verlagert werden können.

Aufbauend auf dieser Modellsprache werden wir das Prinzip der Spracheinbettung mittels struktureller Typanalyse und partieller Auswertung am Beispiel einer sehr einfachen Gastsprache demonstrieren und untersuchen, ob unsere Methode zu effizientem Programmcode führt, inwieweit sie der Gastsprache Beschränkungen auferlegt und wie man die Modellsprache erweitert, um sie als Hostsprache für nahezu beliebige Gastsprachen auszubauen.

Zum Nachweis unserer dritten Hypothese, werden wir versuchen, unsere Modellsprache in Beziehung zu den Phänomenen der C++ -Template-Metaprogrammierung zu setzen. Es wird sich herausstellen, daß dies möglich ist und somit erstmalig ein Modell für diese Programmiertechnik zur Verfügung steht.

Um die Praxistauglichkeit unserer Methode zur Spracheinbettung auf die Probe zu stellen, werden wir schließlich untersuchen, ob sich eine berechnugsvollständige funktionale Sprache mit verzögerter Auswertung und algebraischen Datentypen in C++ integrieren läßt.

1.4 Aufbau dieser Arbeit

Diese Arbeit besteht aus zwei Teilen. Im ersten Teil besprechen wir die Grundlagen der Sprachintegration durch strukturelle Typanalyse und partielle Auswertung.

In Kapitel 2 präsentieren wir die notwendigen Grundlagen. Neben Grundbegriffen aus der universellen Algebra und der Ordnungstheorie werden diverse Varianten des λ-Kalküls vorgestellt und es wird das Prinzip der partiellen Auswertung erklärt. Kapitel 3 befaßt sich mit der Erweiterung des λ-Kalküls um Sprachmittel zur strukturellen Typanalyse, zeigt Beispiele für deren

Anwendung und stellt einen partiellen Auswerter vor, mit dem typabhängige Operationen in die Übersetzungszeit verlagert werden können. Den Kern des ersten Teils dieser Arbeit bildet Kapitel 4. Hier erläutern wir das Prinzip der Sprachintegration durch strukturelle Typanalyse und partielle Auswertung, gehen auf die Grenzen dieses Ansatzes ein und zeigen, wie unsere Modellsprache erweitert werden kann, um diese Grenzen zu sprengen und welche Auswirkungen die Erweiterungen auf die Eigenschaften unserer Modellsprache haben. Darüber hinaus zeigen wir an Beispielen, daß sich unser Ansatz grundsätzlich zur Implementierung von domänenspezifischen Programmtransformationen (Optimierungen) eignet. Im Ergebnis erhalten wir am Ende des Kapitels eine Hierarchie von Sprachen, die die Einbettung von Gastsprachen mit unterschiedlichen Eigenschaften erlauben.

Kapitel 5 bildet das Bindeglied zwischen dem (eher theoretischen) ersten und dem praktischen zweiten Teil dieser Arbeit. Nach einer kurzen Einführung in C++ -Templates und Überladung in C++ werden wir zeigen, wie sich die Sprachelemente unserer Modellsprache auf C++ abbilden lassen. Wir werden feststellen, daß dies für nahezu alle Sprachelemente der Modellsprache möglich ist und sich C++ daher als gute Testplattform anbietet, um die Tauglichkeit unseres Ansatzes an einem Praxisbeispiel zu erproben.

Im zweiten Teil der Arbeit beschäftigen wir uns mit einer mittelgroßen Anwendung der Spracheinbettung durch strukturelle Typanalyse und partielle Auswertung. Konkret werden wir beschreiben, wie sich eine funktionale Sprache mit verzögerter Auswertung und algebraischen Datentypen in C++ einbetten läßt.

In Kapitel 6 starten wir mit einer allgemeinen Motivation und Definition der Multiparadigma-Programmierung, um aufzuzeigen, daß sie besonders hohe Anforderungen an ein Werkzeug zur Spracheinbettung stellt. Insbesondere werden für das objektorientierte und das funktionale Paradigma essentielle Begriffe und Eigenschaften diskutiert, auf die wir im weiteren Verlauf zurückgreifen.

In Kapitel 7 stellen wir die funktionale Sprache EML zunächst anhand von Beispielen vor. Die Syntax von EML wird in Kapitel 8 formal definiert, um anschließend zu zeigen, wie sich EML (syntaktisch) in C++ integrieren läßt. Einen Typechecker für EML entwickeln wir in Kapitel 9, wobei wir auf die Technik der abstrakten Interpretation zurückgreifen. Nachdem der abstrakte Interpreter formal beschrieben wurde, gehen wir auf seine Implementierung in C++ ein.

Die EML zu Grunde liegende Auswertungsstrategie (Graphreduktion) wird in Kapitel 10 zunächst informell beschrieben. Anschließend beschreiben wir eine abstrakte Maschine (einen Interpreter), der EML-Programme mittels Graphreduktion auswertet und besprechen, wie sich die Komponenten dieser Maschine in C++ darstellen lassen. Nachdem die Optimierung von endrekursiven Aufrufen in EML besprochen wurde, werden erste Testergebnisse präsentiert und analysiert. Kapitel 11 beschäftigt sich schließlich mit der Integration von algebraischen Datentypen in EML. Wir werden zeigen, wie sich algebraische Datentypen im Typsystem von C++ darstellen lassen und wie der Typechecker und der Interpreter für EML zu erweitern sind, um algebraische Datentypen auch in EML zugänglich zu machen. Das Kapitel schließt mit der Präsentation und Diskussion von Laufzeitergebnissen.

In Kapitel 12 fassen wir unsere Ergebnisse zusammen und prüfen, inwieweit sich unsere Hypothesen bestätigt haben. Darüber hinaus geben wir einen Ausblick auf mögliche Weiterentwicklungen und Anwendungen.

Teil I

Grundlagen der Sprachintegration durch strukturelle Typanalyse

Kapitel 2

Allgemeine Grundlagen

In diesem Kapitel werden theoretische Grundlagen präsentiert, auf die im weiteren Verlauf der Arbeit zurückgegriffen wird. Neben einer kurzen Einführung von Begriffen aus der universellen Algebra und der Ordnungstheorie, werden diverse Varianten des λ-Kalküls kurz vorgestellt. Darüber hinaus wird in einem kurzen Exkurs das Prinzip der partiellen Auswertung erklärt.

Dieses Kapitel erhebt nicht den Anspruch einer umfassenden theoretischen Modellierung und Diskussion. Vielmehr dient es der Festlegung von Notationen und der Wiederholung der operationellen Semantik und der Durchleuchtung verschiedener Aspekte von Typsystemen.

Am Ende des Kapitels geben wir Hinweise auf vertiefende Literatur.

2.1 Universelle Algebra

Definition 2.1 (Signatur)**.** Sei F eine Menge von Funktionssymbolen und $\sigma : F \to \mathbb{N}$ eine Funktion, die jedem Funktionssymbol eine Stelligkeit zuordnet. Dann heißt $\Sigma = \langle F, \sigma \rangle$ eine **Signatur**. Für $i \in \mathbb{N}$ sei

$$F^i = \{f \in F | \sigma(f) = i\}$$

die Menge der i-stelligen Funktionssymbole. Die Elemente aus F^0 werden auch Konstanten genannt. ◊

Definition 2.2 (Algebra)**.** Sei $\Sigma = \langle F, \sigma \rangle$ eine Signatur, A eine Menge und α eine Abbildung, die jedem Funktionssymbol aus $f \in F^n$ eine Funktion $A^n \to A$ zuordnet. Dann ist $\mathcal{A} = \langle A, \alpha \rangle$ eine **Σ-Algebra**. Anstelle von $\alpha(f)$ schreiben wir auch $f_{\mathcal{A}}$.

◊

Die Menge A kann durchaus aus komplexen Objekten (etwa Termen) bestehen. Es ist wichtig, sich klar zu machen, daß in der Definition einer Algebra keinerlei Aussage darüber gemacht wird, wie die Objekte aus A „hingeschrieben" werden können - geschweige denn, was diese überhaupt sind. Die konkrete Syntax, in der Objekte der Menge A notiert werden, ist zunächst offen.

Definition 2.3 (Homomorphismus)**.** Seien $\mathcal{A} = \langle A, \alpha \rangle$ und $\mathcal{B} = \langle B, \beta \rangle$ Σ-Algebren. Eine Abbildung $h : A \to B$ heißt **Homomorphismus**, falls

$$h(f_{\mathcal{A}}(a_1, \cdots, a_n)) = f_{\mathcal{B}}(h(a_1), \cdots, h(a_n))$$

$\forall f \in F^n$, $a_i \in A$; $n \in \mathbb{N}$. Besitzt $h : A \to B$ die Homomorphieeigenschaft, so schreibt man auch $h : \mathcal{A} \to \mathcal{B}$. $\Diamond$

Anmerkung 2.1. Homomorphismen sind strukturerhaltende Abbildungen zwischen Σ-Algebren. Offensichtlich sind Homomorphismen unter Komposition abgeschlossen; sind also $f : B \to C$ und $g : A \to B$ Homomorphismen, so auch $(f \circ g)(a_1, \cdots, a_n) = f(g(a_1, \cdots, a_n))$. $\Diamond$

Definition 2.4 (Σ-Termalgebra)**.** Sei $\Sigma = \langle F, \sigma \rangle$ eine Signatur und X eine Menge von Variablen mit $F \cap X = \emptyset$. Dann ist die **Σ-Termalgebra** über X

$$\mathcal{T}_\Sigma(X) = \langle T_\Sigma(X), \alpha \rangle$$

definiert durch

1. $T_\Sigma(X)$ ist die kleinste Menge mit
   (a) $X \cup F^0 \subseteq T_\Sigma(X)$ (Variablen und Konstanten)
   (b) $t_1, \cdots, t_n \in T_\Sigma(X)$, $n \geq 1$, $f \in F^n \Rightarrow f(t_1, \cdots, t_n) \in T_\Sigma(X)$ (Applikation)
2. Jedes $f \in F$ wird von α als „Konstruktorfunktion" interpretiert:
   $\alpha(f) : T_\Sigma(X)^n \to T_\Sigma(X)$
   $\alpha(f)(t_1, \cdots, t_n) = f(t_1, \cdots, t_n)$, $\forall f \in F^n, n \in \mathbb{N}, t_i \in T_\Sigma(X)$. $\Diamond$

Definition 2.5 (Frei erzeugte Σ-Algebra)**.** Sei $\Sigma = \langle F, \sigma \rangle$ eine Signatur, $\mathcal{A} = \langle A, \alpha \rangle$ eine Σ-Algebra und $Z \subseteq A$. Dann heißt $\mathcal{A}$ freie Σ-Algebra mit Erzeugermenge Z, falls gilt:

- für jede Σ-Algebra $\mathcal{B} = \langle B, \beta \rangle$ und jede Abbildung $\varphi : Z \to B$ gibt es genau eine homomorphe Erweiterung $\hat{\varphi} : \mathcal{A} \to \mathcal{B}$. $\Diamond$

Satz 2.1. Sei $\Sigma = \langle F, \sigma \rangle$ eine Signatur. Dann ist die Σ-Termalgebra $\mathcal{T}_\Sigma(X) = \langle T_\Sigma(X), \alpha \rangle$ frei erzeugte Σ-Algebra mit Erzeugermenge X.

Da $\mathcal{T}_\Sigma(X)$ frei von X erzeugt ist, kann man diese Σ-Algebra als Stellvertreter einer Klasse von Σ-Algebren betrachten; insbesondere auch deshalb, weil $X = \emptyset$ nicht ausgeschlossen wurde. Der Vorteil der Termalgebra liegt darin, daß man mit ihrer Hilfe von der konkreten Syntax abstrahieren kann.

Definition 2.6 (Die Funktionen $vars$, $fsyms$ und $kopf$)**.** Sei $\Sigma = \langle F, \sigma \rangle$ eine Signatur. Die Funktion $vars : T_\Sigma(X) \to \mathcal{P}(X)$[1] ermittelt die in einem Term vorkommenden Variablen wie folgt:

- $\forall x \in X \ : \ vars(x) = \{x\}$
- $\forall f \in F^0 \ : \ vars(f) = \emptyset$

[1] $\mathcal{P}(x)$ bezeichnet die Potenzmenge von X.

- $vars(f(t_1, \cdots, t_n)) = vars(t_1) \cup \cdots \cup vars(t_n)$ mit $f \in F$, $t_i \in T_\Sigma(X)$ $(1 \leq i \leq n)$

Die Funktion $fsyms : T_\Sigma(X) \to \mathcal{P}(F)$ ermittelt die in einem Term enthaltenen Funktionssymbole:

- $\forall x \in X \ : \ fsyms(x) = \emptyset$
- $\forall f \in F^0 \ : \ fsyms(f) = \{f\}$
- $fsyms(f(t_1, \cdots, t_n)) = \{f\} \cup fsyms(t_1) \cup \cdots \cup fsyms(t_n)$ mit $f \in F$, $t_i \in T_\Sigma(X)$ $(1 \leq i \leq n)$

Die Funktion $kopf : T_\Sigma(X) \to F$ zur Bestimmung des Termkopfes, ist folgendermaßen definiert:

- $\forall f \in F^0 \ : \ kopf(f) = f$
- $kopf(f(t_1, \cdots, t_n)) = f$ mit $f \in F$, $t_i \in T_\Sigma(X)$ $(1 \leq i \leq n)$ ◊

Definition 2.7 (Substitution)**.** Sei $\Sigma = \langle F, \sigma \rangle$ eine Signatur, X eine Variablenmenge und $subst : T_\Sigma(X) \to T_\Sigma(X)$ ein Homomorphismus. Der Homomorphismus *subst* heißt **Substitution**, falls

$$dom(subst) = \{x \in X \mid subst(x) \neq x\}$$

endlich ist. Die Menge aller Substitutionen wird mit **Sub** bezeichnet. ◊

Substitutionen werden wir oft in der Form $subst = \{x_1 \mapsto s_1, \cdots, x_n \mapsto s_n\}$ angeben und meinen damit, daß $subst(x_1) = s_1$, $subst(x_2) = s_2$ usw. gilt.

Anmerkung 2.2. Da $\mathcal{T}_\Sigma(X)$ frei von X erzeugte Σ-Algebra ist, ist ein Homomorphismus *subst* bereits eindeutig bestimmt, wenn man eine Belegung der Variablen angibt, also eine Funktion $subst' : X \to T_\Sigma(X)$ vorgibt. Da Homomorphismen unter Komposition abgeschlossen sind, führt die Komposition zweier Substitutionen wieder zu einer Substitution. ◊

Definition 2.8 (Unifikation)**.** Sei $\mathcal{T}_\Sigma(X) = \langle T_\Sigma(X), \alpha \rangle$ eine Σ-Termalgebra und $t_1, t_2 \in T_\Sigma(X)$. Eine Substitution $subst \in$ **Sub** heißt **Unifikator** für t_1 und t_2, falls $subst(t_1) = subst(t_2)$ gilt. Ein Unifikator *subst* heißt allgemeinster Unifikator, falls für jeden Unifikator *subst'* ∈ **Sub** gilt: $\exists \lambda \in$ **Sub** $: subst' = \lambda \circ subst$. ◊

Terme kann man sich anschaulich auch als Bäume vorstellen (siehe Abbildung 2.1). Mit Hilfe der Standardnumerierung von Bäumen kann man Stellen im Term gezielt ansprechen und damit die Begriffe Unterterm und Untertermsubstitution einführen.

Definition 2.9 (Unterterm / Untertermsubstitution)**.** Die Menge $O(t)$ der Positionen (*occurrences*) eines Terms ist eine Menge von Wörtern über dem Alphabet der natürlichen Zahlen und dem Symbol „.", die induktiv definiert ist durch[2]

[2] ϵ bezeichne das leere Wort.

- $O(f) = \{\varepsilon\}$ für $f \in F^0$
- $O(x) = \{\varepsilon\}$ für $x \in X$
- $O(t) = \{\varepsilon\} \cup \{i.p \,|\, 0 \leq i \leq n, p \in O(t_i), p \neq \epsilon\} \cup \{0..n\}$ falls $t = f(t_0, \cdots, t_n)$.

Ein **Unterterm** $t|_p$ von $t = f(t_0, \cdots, t_n)$ an der Stelle $p \in O(t)$ ist definiert durch:

- $t|_p = t$, falls $p = \epsilon$
- $t|_p = t_i$, falls $p \in \mathbb{N}$
- $t|_p = t_i|_q$, falls $p = i.q$, $0 \leq i \leq n$.

Mit $t[u \leftarrow s]$ wird der Term bezeichnet, den man aus t erhält, wenn man den Unterterm von t an der Stelle u ($t|_u$) durch s ersetzt.
Die Funktion $depth_t(u)$, $u \in O(t)$, $t \in T_\Sigma(X)$, die Tiefe des Teilterms von t an der Stelle u, ist induktiv definiert durch

- $depth_t(\varepsilon) = 0$
- $depth_t(i) = 1$, falls $i \in \mathbb{N}$
- $depth_t(i.q) = 1 + depth_t(q)$, falls $i \in \mathbb{N}, q \in O(t)$. ◊

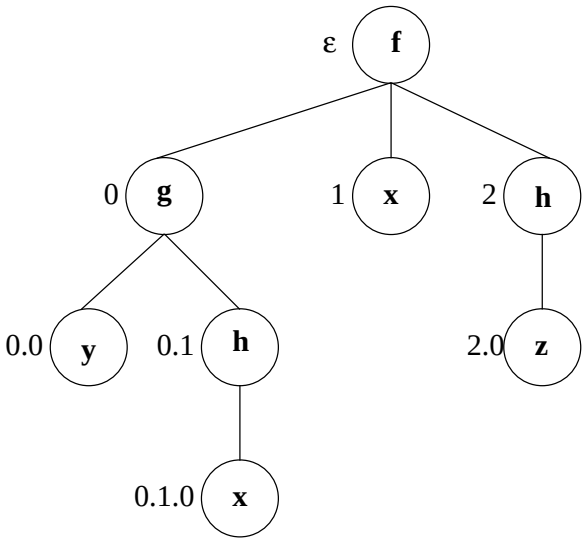


Abbildung 2.1: Standardnumerierung von Bäumen am Beispiel des Terms $t = f(g(y, h(x)), x, h(z))$.

Anmerkung 2.3. $depth_t(u)$ zählt gewissermaßen die Zahlen, die in einer Positionsmarke auftauchen. So ist zum Beispiel $depth_t(0.1.0) = 3$. ◊

Definition 2.10 (Match / Matchordnung)**.** Seien $\mathcal{T}_\Sigma(X) = \langle T_\Sigma(X), \alpha\rangle$ eine Σ-Termalgebra und $s, t \in T_\Sigma(X)$ Terme. Dann sind die Relationen $\trianglelefteq$ und $\equiv$ über $T_\Sigma(X) \times T_\Sigma(X)$ definiert durch:

- $s \trianglelefteq t \;\Leftrightarrow\; \exists subst \in \mathbf{Sub} : subst(s) = t$. *subst* heißt dann **Match** von s auf t und $\trianglelefteq$ heißt **Matchordnung**.
- $s \equiv t \;\Leftrightarrow\; s \trianglelefteq t \wedge t \trianglelefteq s$. ◊

Mit Hilfe der Relation $\trianglelefteq$ lassen sich Terme hinsichtlich ihrer Allgemeinheit sortieren. Gilt $s \trianglelefteq t$, so ist s **allgemeiner** als t bzw. t **spezieller** als s. Ohne Beweis sei bemerkt, daß $\trianglelefteq$ eine Vorordnung (also reflexiv und transitiv) ist und $\equiv$ eine Äquivalenzrelation (also reflexiv, transitiv und symmetrisch) ist. Eine Äquivalenzklasse von $\equiv$ wird mit $[t]_\equiv = \{t' | t' \equiv t\}$ bezeichnet. Zwei Terme sind also genau dann äquivalent, wenn sie bis auf Umbenennung der Variablen identisch sind.

2.2 Ordnungstheoretische Grundlagen

Definition 2.11 (Halbordnung)**.** Sei A eine Menge und $\leq \subseteq A \times A$ eine Relation, so daß $\leq$

- reflexiv: $a \leq a$
- transitiv: $a \leq b \wedge b \leq c \Rightarrow a \leq c$
- anti-symmetrisch: $a \leq b \wedge b \leq a \Rightarrow a = b$

für alle $a, b \in A$. Dann heißt $\mathcal{A} = \langle A; \leq\rangle$ eine **Halbordnung**. ◊

Definition 2.12 (Monotone Funktion)**.** Seien $\mathcal{A} = \langle A; \leq_A\rangle$ und $\mathcal{B} = \langle B; \leq_B\rangle$ Halbordnungen und $f : A \to B$ eine Funktion. f heißt **monoton**, falls

$$\forall a_1, a_2 \in A \text{ mit } a_1 \leq_A a_2 \Rightarrow f(a_1) \leq_B f(a_2). \qquad \text{◊}$$

Korollar 2.1. Monotone Funktionen sind unter Komposition abgeschlossen.

Definition 2.13 (Gerichtete Menge)**.** Sei $\mathcal{A} = \langle A; \leq\rangle$ eine Halbordnung und $T \subseteq A$; $T \neq \emptyset$. Dann heißt T gerichtet genau dann, wenn

$$\forall a_1, a_2 \in T \, \exists b \in T : a_1 \leq b \wedge a_2 \leq b. \qquad \text{◊}$$

Anmerkung 2.4 (Mengenschreibweise). Anstatt $\forall t \in T : a \leq t$ schreibe: $a \leq T$; anstelle von $\forall t \in T : t \leq a$ kurz: $T \leq a$ und für $\{f(t) | t \in T\}$ einfach: $f(T)$. ◊

Definition 2.14 (Obere Schranke, kleinstes Element, Supremum)**.** Sei $\mathcal{A} = \langle A; \leq\rangle$ eine Halbordnung, $T \subseteq A$ und $T \neq \emptyset$. Dann heißt $a \in A$ **obere Schranke** von T, falls $T \leq a$. a heißt **kleinstes Element** von T, falls $a \leq T$ und $a \in T$. Besitzt $\{b \,|\, T \leq b\}$ ein kleinstes Element, so heißt dieses kleinste obere Schranke oder **Supremum** $\bigsqcup T$ von T. ◊

Definition 2.15 (Untere Schranke, größtes Element, Infimum)**.** Sei $\mathcal{A} = \langle A; \leq \rangle$ eine Halbordnung, $T \subseteq A$ und $T \neq \emptyset$. Dann heißt $a \in A$ **untere Schranke** von T, falls $a \leq T$. a heißt **größtes Element** von T, falls $T \leq a$ und $a \in T$. Besitzt $\{b \mid b \leq T\}$ ein größtes Element, so heißt dieses größte untere Schranke oder **Infimum** $\bigsqcap T$ von T. ◊

Definition 2.16 (Vollständige Halbordnung)**.** Sei $\mathcal{A} = \langle A; \leq \rangle$ eine Halbordnung. $\mathcal{A}$ heißt vollständig, falls gilt:

1. A besitzt ein kleinstes Element $\bot \in A$
2. Jede gerichtete Teilmenge $T \subseteq A$ hat in A eine kleinste obere Schranke $\bigsqcup T \in A$. ◊

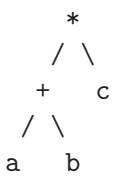
2.3 Ein erweiterter getypter λ-Kalkül: $\lambda^{\mathtt{fix}}$

2.3.1 Syntax

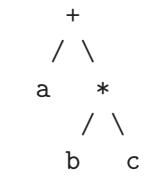
Bei der Beschreibung der Syntax einer Programmiersprache unterscheidet man zwischen abstrakter und konkreter Syntax. Die konkrete Syntax beschreibt, wie Programme in einer Sprache tatsächlich aussehen, also wie ein Programmierer sie eingibt. Unter der abstrakten Syntax versteht man eine erheblich vereinfachte Programmdarstellung in Form eines Baums – dem abstrakten Syntaxbaum (AST). Sie ist das Produkt aus den frühen Analysephasen eines Übersetzers. Ein Programmtext wird zunächst durch die lexikalische Analyse in eine Sequenz von sogenannten **Token** zerlegt. Das sind z.B. Bezeichner, Konstanten, Schlüsselwörter, Kommentare usw. Token, die für die Bedeutung eines Programmes nicht relevant sind (z.B. Kommentare, Leerzeichen, Tabulatoren), werden dabei herausgefiltert.

Anschließend wird der Tokenstrom einem **Parser** zugeführt, der daraus den AST konstruiert. Der Parser berücksichtigt Operatorpräzedenzen und Assoziativitäten und macht diese in der Baumstruktur sichtbar, so daß in der abstrakten Syntax auf Klammern verzichtet werden kann.

Zum Beispiel würde aus dem Term $(a + b) * c$ der Baum



entstehen; wohingegen der Term $a + b * c$ durch diesen AST repräsentiert wird:



(∗ bindet stärker als +)

Die abstrakte Syntax orientiert sich an der Baumdarstellung, weshalb man hier auf die Terminalstrings der konkreten Syntax (z.B. die runden Klammern) verzichten kann. Aus Platzgründen werden abstrakte Syntaxbäume gerne als Zeichenkette notiert. Diese entsteht durch Traversieren des AST mit einer *Präorder*-Strategie; z.B. schreibt man den Term $(a + b) * c$ in konkreter Syntax als $* \ + \ a \ b \ c$ in abstrakter Notation.

In dieser Arbeit werden wir uns vornehmlich auf die abstrakte Syntax einer Sprache konzentrieren. Grammatiken beschreiben daher eher gültige Baumstrukturen als gültige Programme.

Um Programme ästhetischer formulieren zu können, erlauben wir in der abstrakten Notation (als Zeichenkette) die infix-Notation von binären Operatoren, oder setzen zumindest in Beispielen eine konkrete Syntax voraus, die uns syntaktischen Zucker liefert, um die infix-Schreibung von binären Operatoren zu ermöglichen, oder Terme durch Klammerung übersichtlicher darstellen zu können.

Eine klare Abgrenzung von konkreter und abstrakter Syntax wollen wir nicht weiter verfolgen – sie wird sich in der Regel aus dem Kontext heraus ergeben.

Syntax $\lambda^{\texttt{fix}}$

$T ::=$		**Typen**
	`bool`	Wahrheitswerte
	`int`	Ganze Zahlen
	$T \to T$	Funktionstyp$^{(*)}$
$t ::=$		**Terme**
	x	Variable$^{(*)}$
	`if` t `then` t `else` t	Selektion
	$t\{+, -, *, /, ==\}t$	Binäre Operation
	$!\, t$	Logisches nicht
	`fix` t	Fixpunktberechnung
	$\lambda x : T.t$	λ-Abstraktion
	$t\ t$	Applikation$^{(*)}$
$c ::=$		**Konstanten**
	`true`	Konstante „Wahr"
	`false`	Konstante „Falsch"
	$\ldots, -1, 0, 1, \ldots$	Numerische Konstanten
$v ::=$		**Werte**
	c	Konstante
	$\lambda x : T.t$	λ-Abstraktion$^{(*)}$

Abbildung 2.2: Syntax von $\lambda^{\texttt{fix}}$. Syntaktische Formen, die den λ-Kalkül ohne Erweiterungen beschreiben, sind mit $^{(*)}$ gekennzeichnet.

Abbildung 2.2 zeigt die Grammatik zur Syntax eines erweiterten λ-Kalküls, den wir $\lambda^{\texttt{fix}}$-Kalkül nennen. Die Grammatik haben wir als Tabelle, in einer leichten Variation der Backus Naur Form (BNF) angegeben.

Die Tabelle ist in Blöcke eingeteilt. Die jeweils erste Zeile (z.B. $t ::=$) legt den zu definierenden syntaktischen Bereich – hier die Menge der Terme – fest und zeichnet die Variable t als Platzhalter für beliebige Terme aus. Jede nachfolgende Zeile beschreibt eine alternative syntaktische Form für Terme, wobei an Stellen, wo die Variable t steht, beliebige Terme eingesetzt werden können. Die Definition ist also rekursiv. In der dritten Spalte geben wir der syntaktischen Form einen Namen.

Aus Platzgründen verwenden wir bei der Definition der binären infix-Operatoren eine Mengenschreibweise. Mit $t\ \{+,-,*,/,==\}\ t$ ist gemeint, daß zwischen den zwei Termen genau eines der aufgeführten Operatorsymbole stehen darf.

Die Variable t wird auch **Metavariable** genannt – *Variable*, weil wir beliebige Terme für sie einsetzen können und *Meta*, weil es sich um keine Variable der Objektsprache (dem $\lambda^{\texttt{fix}}$-Kalkül) handelt.

Taucht eine Metavariable auf der rechten Seite von ::= mehrfach auf (wie z.B. bei der Funktionsapplikation), so bedeutet dies nicht, daß hier identische Terme eingesetzt werden müssen. Wenn es uns wichtig erscheint, die mögliche Verschiedenheit von Termen zu betonen, oder, um die verschiedenen Vorkommen einer Metavariable gezielt ansprechen zu können, werden wir Metavariablen auch mit Indizes versehen. Dann stehen z.B. t_1, t_2 etc. für Terme, T_1, T_2 etc. für Typen und so fort.

Wir unterscheiden Werte und Terme. Die Menge der Werte beschreibt Objekte, die als Ergebnis einer Berechnung hervorgehen können. Streng genommen müssen diese syntaktischen Objekte noch interpretiert werden, und wir müssten zwischen der 1 als syntaktisches Objekt und der 1 als Element der ganzen Zahlen unterscheiden. Dasselbe gilt für Operatorsymbole wie `+`, die einmal als syntaktisches Objekt und ein anderes Mal als mathematische Funktion verwendet werden. In der Regel wird die Bedeutung aus dem Kontext heraus klar. Gegebenfalls überstreichen wir Werte und Operatoren, um deutlich zu machen, daß das interpretierte Objekt gemeint ist. Dann bezeichnet $1+3$ ein syntaktisches Objekt, während $\overline{1+3}$ das mathematische Objekt ist, welches wir durch $\overline{4}$ ersetzen können. Die Rücktransformation zu einem syntaktischen Objekt verstehen wir als rein impliziten Vorgang.

Die Applikation setzen wir als linksassoziativ voraus; die Wirkung von λ-Bindungen erstrecke sich so weit wie möglich nach rechts. Das heißt:

$$\begin{array}{lll} t_1\ t_2\ t_3 & \text{steht für} & ((t_1\ t_2)\ t_3) \\ \lambda x:T.t_1\ t_2 & \text{steht für} & \lambda x:T.(t_1\ t_2) \end{array}$$

2.3.2 Grundlegende Begriffe

Der Geltungsbereich (engl.: *scope*) einer Variablen in einer λ-Abstraktion $\lambda x:T.t$ erstreckt sich immer über den gesamten Term t. Man sagt, daß x in t eine **gebundene Variable** ist. Ist x in t nicht gebunden, so heißt x **freie Variable** in t.

Die Menge $FV(t)$, die alle in t frei vorkommenden Variablen enthält, ist induktiv definiert durch

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(\lambda x:T.t) & = & FV(t)\backslash\{x\} \\ FV(t_1\ t_2) & = & FV(t_1) \cup FV(t_2) \\ FV(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 & = & FV(t_1) \cup FV(t_2) \cup FV(t_3) \\ FV(t_1\ \{+,-,*,/,==\}\ t_2) & = & FV(t_1) \cup FV(t_2) \\ FV(!t) & = & FV(t) \\ FV(\texttt{fix}\ t) & = & FV(t) \end{array}$$

Einen Term, der keine freien Variablen enthält, nennen wir **abgeschlossenen Term** (engl.: *closed term*).

Wir schreiben $t[s/x]$ (lies: s für x in t), um den Term zu beschreiben, der durch Substitution von x mit dem Term s entsteht:

$$\begin{array}{rcll} x[s/x] & = & s & \\ x[s/y] & = & x & \text{falls } x \neq y \\ (\lambda y.t)[s/x] & = & \lambda y.(t[s/x]) & \text{falls } x \neq y \text{ und } y \notin FV(s) \\ (\lambda x.t)[s/x] & = & \lambda x.t & \\ (t_1 t_2)[s/x] & = & t_1[s/x]t_2[s/x] & \\ \text{usw.} & & & \end{array}$$

Mit der Forderung, daß y in s nicht frei vorkommen darf, schließen wir den Fall aus, daß eine Variable in s irrtümlich gebunden wird. Ohne diese Forderung wäre z.B.

$$\lambda y.x\ y[yz/x] = \lambda y.(y\ z)z$$

und die in $y\ z$ frei vorkommende Variable y würde „durch die Hintertür" gebunden.

Wir legen fest, daß zwei Terme, die sich nur in den Namen der gebundenen Variablen unterscheiden, **äquivalente Terme** sind. Die Terme $\lambda x : T.x\ z$ und $\lambda y : T.y\ z$ sind beispielsweise äquivalent (x ist gebunden und wurde zu y umbenannt); die Terme $\lambda x : T.x\ z$ und $\lambda x : T.x\ y$ hingegen nicht.

Die Äquivalenz von Termen können wir ausnutzen, um Konflikte bei der Substitution zu umgehen. Betrachten wir erneut den Problemfall $\lambda y.x\ y[yz/x]$. Dieser λ-Term ist äquivalent zu $\lambda w.x\ w$; wir können also zu $\lambda w.x\ w[x/yz]$ übergehen und damit den Konfliktfall umgehen.

Die Umbenennung von gebundenen Variablen heißt **α-Konvertierung** und fortan gehen wir davon aus, daß - sofern erforderlich - α-Konvertierungen bei der Substitution automatisch durchgeführt werden.

2.3.3 Reduktionsstrategien

In seiner puren Form kennt der λ-Kalkül keine Konstanten und keine eingebauten Primitive. Grundlegend für die Berechnung ist die Anwendung einer Funktion auf ein Argument, welches selbst wieder eine Funktion sein kann.

Ein Berechnungsschritt bedeutet den Übergang von einer Applikation $(\lambda x.t_1)\ t_2$ zu einem Term t_3, wobei man t_3 aus t_1 erhält, indem man alle Vorkommen der gebundenen Variablen x in t_1 durch t_2 ersetzt; bzw. formal:

$$(\lambda x.t_1)\ t_2 \rightarrow t_1[t_2/x]$$

Diesen für den λ-Kalkül essentiellen Reduktionsschritt nennt man **β-Reduktion**.

Bemerkung: $\rightarrow$ heißt Reduktionsrelation und mit $\stackrel{*}{\rightarrow}$ beschreiben wir ihren reflexiven und transitiven Abschluß.

Ein Term, der sich mit Hilfe von β-Reduktion (oder einer der noch vorzustellenden Regeln) reduzieren läßt, ist ein **Redex**. Ein Term t, der nicht weiter reduziert werden kann, heißt **Normalform** und wir schreiben $t \Downarrow$ um dies kenntlich zu machen.

In einem Term kann es mehrere Redexe geben und die Reduktionsrelation $\rightarrow$ läßt einem die freie Wahl, welchen Redex man als nächstes reduziert.

Diese Wahlfreiheit ist insofern unkritisch, als daß die β-Reduktion konfluent ist (siehe z.B. [13, Theorem 2.3.7]). Das heißt, daß sich zwei auseinanderlaufende Reduktionspfade grundsätzlich wieder zusammenführen lassen:

$$t_s \xrightarrow{*} t_1 \wedge t_s \xrightarrow{*} t_2 \Rightarrow \exists t_z : t_1 \xrightarrow{*} t_z \wedge t_2 \xrightarrow{*} t_z$$

Unendliche Reduktionspfade sind dadurch natürlich nicht ausgeschlossen.

Hinsichtlich einer Implementierung ist man mehr an einem deterministischen Verfahren zur Auswertung von $\lambda^{\texttt{fix}}$-Termen interessiert.

Diverse Ansätze mit jeweils unterschiedlichen Eigenschaften sind hierzu untersucht worden. Bei der ***normal order*** Strategie wird immer der am weitesten links außen stehende Term reduziert; z.B.:

$$\begin{aligned} & (\lambda x : \texttt{int}.x + x)(3 * 3) \\ \rightarrow \quad & 3 * 3 + 3 * 3 \\ \rightarrow \quad & 9 + 3 * 3 \\ \rightarrow \quad & 9 + 9 \\ \rightarrow \quad & 18 \end{aligned}$$

Offenbar wird der Term $3 * 3$ hier zweimal ausgewertet – später werden wir Varianten der *normal order* Strategie kennenlernen, bei denen die doppelte Argumentauswertung verhindert werden kann.

Eine interessante Eigenschaft der *normal order reduction* ist, daß diese Strategie immer zu einer Normalform führt – sofern diese existiert (siehe [12, Theorem 13.2,2]).

Eine Auswertungsstrategie, die sich leichter auf von Neumann Rechner abbilden läßt, ist die ***call by value*** Strategie, bei der Funktionsargumente ausgewertet werden, bevor sie an eine Funktion übergeben werden:

$$\begin{aligned} & (\lambda x : \texttt{int}.x + x)(\ 3 * 3\ ) \\ \rightarrow \quad & (\lambda x : \texttt{int}.x + x)(9) \\ \rightarrow \quad & 9 + 9 \\ \rightarrow \quad & 18 \end{aligned}$$

2.3.4 Operationelle Semantik

Mit einer Semantik wird den syntaktischen Formen einer Sprache eine Bedeutung zugeordnet. Zur Beschreibung der Semantik von $\lambda^{\texttt{fix}}$-Termen werden wir auf einen operationellen Ansatz zurückgreifen, da dieser näher an einer Implementierung ist, als beispielsweise eine denotationelle Semantik, die sich primär mit semantischen Bereichen und der Interpretation durch Funktionen beschäftigt.

Eine operationelle Semantik für $\lambda^{\texttt{fix}}$ beschreibt die Bedeutung eines Terms durch eine abstrakte Maschine, deren Zustand ein einfacher $\lambda^{\texttt{fix}}$-Term ist. Mögliche Zustandsübergänge (Transitionen) beschreiben wir durch Regeln der Form:

$$\frac{\text{Prämisse(n)}}{t_1 \rightarrow t_2} \quad \textbf{(E-Regelname)}$$

Diese Regel mit Namen **E-Regelname**[3] sagt aus, daß die Maschine unter der Voraussetzung, daß die in der Prämisse genannten Übergänge möglich sind (bzw. die in der Prämisse geforderten Bedingungen erfüllbar sind), vom Zustand t_1 zum Zustand t_2 übergehen kann. Mit anderen Worten kann der Term t_1 mit einem Maschinenschritt zum Term t_2 reduziert werden.

Ein Endzustand ist erreicht, wenn der aktuelle Zustand ein Wert ist.

Abbildung 2.3 zeigt das vollständige Regelwerk zur Beschreibung aller möglichen Zustandsübergänge. Wie gehabt, bezeichnen wir mit $t[v/x]$ den Term, den wir aus t gewinnen, indem wir alle Vorkommen von x in t durch v ersetzen (z.B. ist $(x+2)[0/x] = 0+2$). Wollen wir in einem Term gleich mehrere Variable ersetzen, schreiben wir dies als $t[t_1/x_1, \cdots, t_n/x_n]$.

Grundsätzlich kann eine Regel dann auf einen Term angewendet werden, wenn es eine Substitution für die Metavariablen der linken Seite des Zustandsübergangs gibt, die diese mit dem Term unifiziert. Dabei dürfen Metavariablen natürlich nur durch syntaktische Formen aus dem ihnen zugeordneten Bereich ersetzt werden.

Zum Beispiel kann der Automat aus dem Zustand $(\lambda x : \texttt{int}.x + 3)\ 4$ mit Regel **E-AppAbs** in den Zustand $4 + 3$ übergehen. Die Regel greift, da man mit $\lambda x : T.t[\texttt{int}/T,\ x + 3/t]$ eine geeignete Substitution finden kann.

Komplexer wird die Anwendung von Regeln, die Prämissen enthalten. Nehmen wir zum Beispiel den Term $7 + (\lambda x : \texttt{int}.x + 3)\ 4$, so ist Regel **E-+-2** anwendbar, wenn man v_a durch den Wert 4 und t_b durch den Term $(\lambda x : \texttt{int}.x + 3)\ 4$ ersetzt. Allerdings ist ein Übergang nur dann erlaubt, wenn sich t_b zu einem Term t_b' reduzieren läßt. Wir haben bereits gezeigt, daß dies möglich ist und dabei der Term $4 + 3$ entsteht. Der Automat kann somit folgenden Zustandsübergang vollziehen:

$$7 + (\lambda x : \texttt{int}.x + 3)\ 4 \to 7 + (4 + 3) \quad \textbf{(E-+-2)}$$

Im nächsten Schritt kann abermals die Regel **E-+-2** angewendet werden – diesmal entspricht t_b dem Term $4 + 3$ und aus dem Zustand $4 + 3$ kann der Automat mit Regel **E-+-C** in den Zustand **7** übergehen, was dann den Übergang

$$7 + (4 + 3) \to 7 + 7 \quad \textbf{(E-+-2)}$$

erlaubt. Nochmaliges Anwenden von **E-+-C** führt den Automaten schließlich in den Endzustand 14. Derartige Ableitungen kann man sehr gut als Baum darstellen:

$$\dfrac{\dfrac{\dfrac{(\lambda x : \texttt{int}.x + 3)\ 4 \to 4 + 3 \quad \textbf{(E-AppAbs)}}{7 + (\lambda x : \texttt{int}.x + 3)\ \to 7 + (4 + 3)}\ \textbf{(E-+-2)} \quad 4 + 3 \to 7 \quad \textbf{(E-+-C)}}{7 + (4 + 3) \to 7 + 7}\ \textbf{(E-+-2)}}{7 + 7 \to 14}\ \textbf{(E-+-C)}$$

Die Reihenfolge der Auswertung können wir durch geschickte Wahl der Metavariablen steuern: Da v sich ausschließlich über Werte erstreckt, kann die Regel **E-AppAbs** nur angewendet werden, wenn die rechte Seite (das Argument) einer Applikation eine Variable ist. **E-App1** kann hingegen nur dann angewendet werden, wenn t_1 kein Wert ist – insbesondere weil in der Prämisse gefordert wird, daß t_1 zu t_1' reduzierbar ist – eine Bedingung, die z.B. von

[3] „**E**" steht für *Evaluation*

$\boxed{t_1 \to t_2}$ $\lambda^{\texttt{fix}}$

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}\ \textbf{(E-App1)} \qquad \frac{t \to t'}{vt \to vt'}\ \textbf{(E-App2)}$$

$$(\lambda x : T.t)\ v \to t[x/v]\ \textbf{(E-AppAbs)}$$

$$\frac{t_c \to t_c'}{\text{if } t_c \text{ then } t_t \text{ else } t_e \to \text{if } t_c' \text{ then } t_t \text{ else } t_e}\ \textbf{(E-IfStep)}$$

$$\text{if } \texttt{true} \text{ then } t_t \text{ else } t_e \to t_t\ \textbf{(E-IfTrue)} \qquad \text{if } \texttt{false} \text{ then } t_t \text{ else } t_e \to t_e\ \textbf{(E-IfFalse)}$$

$$\frac{t_a \to t_a'}{t_a + t_b \to t_a' + t_b}\ \textbf{(E-+-1)} \qquad \frac{t_b \to t_b'}{v_a + t_b \to v_a + t_b'}\ \textbf{(E-+-2)}$$

$$v_a + v_b \to \overline{v_a + v_b}\ \textbf{(E-+-C)}$$

Analog für $\{-, *, /, ==\}$, wobei $\overline{v_a + v_b}$ das Ergebnis der zugehörigen arithmetischen Operation bzw. der entsprechenden Vergleichsoperation ist.

$$\frac{t \to t'}{!t \to !t'}\ \textbf{(E-Not-Step)}$$

$$!\texttt{false} \to \texttt{true}\ \textbf{(E-Not-False)} \qquad !\texttt{true} \to \texttt{false}\ \textbf{(E-Not-True)}$$

$$\frac{t \to t'}{\text{fix } t \to \text{fix } t'}\ \textbf{(E-Fix)} \qquad \text{fix } \lambda f : T.t \to t[\text{fix } \lambda f : T.t/f]\ \textbf{(E-FixBeta)}$$

Abbildung 2.3: Operationelle *call by value* Semantik von $\lambda^{\texttt{fix}}$.

Konstanten nicht erfüllt wird. Im Vergleich dazu greift **E-App2** nur dann, wenn die linke Seite einer Applikation (die Funktion) ein Wert v ist.

Das Regelwerk impliziert somit eine klare Reihenfolge zur Auswertung eines Terms $t_1\ t_2$: Zuerst wird t_1 zu einem Wert reduziert; anschließend das Argument t_2, um dann mit der Regel **E-AppAbs** die eigentliche Applikation durchzuführen – dieses Vorgehen entspricht exakt der *call by value* Strategie.

Wir haben zur Festlegung der Semantik einen *small-step*-Stil verwendet. *Small-step*, da eine Reduktionsregel genau einen Rechenschritt beschreibt. Im Gegensatz dazu beschreibt eine *big-step* Semantik die Reduktion eines Terms t zu einer Normalform v in einem Schritt (i.Z. $t \Downarrow v$) – eine Reduktionsregel umfaßt hier mehrere atomare Rechenschritte.

Der Vorteil einer *big-step* Semantik liegt in einer kompakteren Formulierung des Regelwerks. Zum Beispiel würde man zur Beschreibung der Addition mit nur einer Regel auskommen (anstelle von drei Regeln bei der *small-step*-Semantik):

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2}{t_1 + t_2 = \overline{v_1 + v_2}}\ \textbf{(BS–E-Plus)}$$

Wenn es uns bequemer erscheint, werden wir später auch auf den *big-step*-Stil zurückgreifen.

2.3.5 Typsystem

Die soeben definierte Semantik läßt einige Terme unberücksichtigt. Zum Beispiel gibt es keine Regel zur Reduktion des syntaktisch korrekten Terms `false` + 4, da die Addition einer booleschen- und einer integer-Konstante nicht definiert ist. Als Konsequenz würde der Automat zur Termberechnung keinen Berechnungsschritt vollziehen können und stehenbleiben, ohne einen gültigen Endzustand erreicht zu haben. Fehler, die auf die Inkompatibilität von erwartetem und tatsächlichem Argumenttyp zurückzuführen sind, nennt man allgemein **Typfehler**.

Ein Typsystem versucht derart fragwürdige Terme auszufiltern, indem es die Zahl der gültigen $\lambda^{\texttt{fix}}$-Terme auf solche einschränkt, denen mit Hilfe des Typsystems ein Typ zugeordnet werden kann. Ziel ist, syntaktisch korrekte, aber logisch unsinnige Terme aus der Sprache herauszufiltern.

Eine **Typzuweisung** (oder auch **Typisierung**) der Form

$$\Gamma \vdash v : T$$

sagt aus, daß das syntaktische Objekt v unter der **Typisierungsumgebung** Γ den Typ T hat.

Eine Typisierungsumgebung ist eine partielle Abbildung von der Menge der Variablen des $\lambda^{\texttt{fix}}$-Kalküls in die Menge der Typen. In ihr werden die Typen der freien Variablen eines Terms vermerkt.

Mit $dom(\Gamma)$ bezeichnen wir den Definitionsbereich von Γ und anstelle von $\Gamma(v) = T$ schreiben wir $v : T \in \Gamma$.

Eine Umgebung kann mit Hilfe des Komma-Operators erweitert werden. So steht $\Gamma, x : T$ für eine Umgebung, die der Variablen x den Typ T zuordnet. Um Doppeldeutigkeiten zu vermeiden, gehen wir davon aus, daß x so gewählt wurde, daß es in $dom(\Gamma)$ noch nicht enthalten ist - dieses Vorgehen ist Ausdruck unserer Konvention, gebundene Variablen bei Bedarf umzubenennen.

Für die leere Umgebung verwenden wir das Symbol $\emptyset_\Gamma$ und anstelle von $\emptyset_\Gamma \vdash \cdots$ schreiben wir kurz $\vdash \cdots$.

Ein Typinferenzsystem für $\lambda^{\texttt{fix}}$ beschreiben wir durch Regeln der Form:

$$\frac{\text{Prämisse(n)}}{\Gamma \vdash t : T} \quad \textbf{(T-Regelname)}$$

Diese Regel mit Namen **T-Regelname**[4] beschreibt, daß wir unter der Voraussetzung, daß die Annahmen der Prämisse(n) korrekt sind, folgern können, daß der Term t unter der Typisierungsumgebung Γ vom Typ T ist. Zum Beispiel drückt die Regel **T-Var** aus, daß wir für den Fall, daß die Typisierungsumgebung Γ eine Typzuweisung $x : T$ enthält, davon ausgehen können, daß die Variable x den Typ T hat.

Typherleitungen können wir ebenfalls als Baum darstellen, was wir am Beispiel des Terms $(\lambda x : \texttt{int}.x)\ 3$ kurz zeigen wollen:

[4] „**T**" steht für *Type*

$\boxed{\Gamma \vdash t : T}$ $\lambda^{\texttt{fix}}$

$$\vdash \text{false} : \texttt{bool} \quad \textbf{(T-False)} \qquad \vdash \text{true} : \texttt{bool} \quad \textbf{(T-True)}$$

$$\vdash \text{z} : \texttt{int} \quad \textbf{(T-Int)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \textbf{(T-Var)}$$

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T.t : T \to T'} \quad \textbf{(T-Abs)} \qquad \frac{\Gamma \vdash t_1 : T_1 \to T_1' \;\; \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1(t_2) : T_1'} \quad \textbf{(T-App)}$$

$$\frac{\Gamma \vdash t_c : \texttt{bool} \;\; \Gamma \vdash t_t : T \;\; \Gamma \vdash t_e : T}{\Gamma \vdash \text{if } t_c \text{ then } t_t \text{ else } t_e : T} \quad \textbf{(T-If)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{int} \quad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 \; \eta \; t_2 : \texttt{int}} \quad \textbf{(T-BinOp)}$$

Mit $\eta \in \{+, -, *, /\}$ und $z \in \mathbb{Z}$.

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 == t_2 : \texttt{bool}} \quad \textbf{(T-Equal)} \qquad \frac{\Gamma \vdash t : \texttt{bool}}{\Gamma \vdash !t : \texttt{bool}} \quad \textbf{(T-Not)}$$

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \text{fix } t : T} \quad \textbf{(T-Fix)}$$

Abbildung 2.4: Typsystem zu $\lambda^{\texttt{fix}}$.

$$\dfrac{\dfrac{\dfrac{x : \texttt{int} \in \emptyset_\Gamma, x : \texttt{int}}{\emptyset_\Gamma \vdash x : \texttt{int}} \; \textbf{(T-Var)}}{\vdash \lambda x : \texttt{int}.x : \texttt{int} \to \texttt{int}} \; \textbf{(T-Abs)} \qquad \dfrac{}{\vdash 3 : \texttt{int}} \; \textbf{(T-int)}}{\vdash (\lambda x : \texttt{int}.x) \; 3 : \texttt{int}} \; \textbf{(T-App)}$$

Wie man leicht nachvollzieht, kann für den Term $(\lambda x : \texttt{int}.x)$ `true` kein Typ inferiert werden, da die Regel **T-App** in diesem Fall nicht anwendbar ist. Es handelt sich folglich um keinen gültigen $\lambda^{\texttt{fix}}$-Term.

2.3.6 Eigenschaften von $\lambda^{\texttt{fix}}$

Für alle abgeschlossenen typisierbaren Terme gilt, daß sie entweder reduzierbar oder aber ein Wert sind:

Fortschritt	$\Gamma \vdash t : T \;\; \Rightarrow \;\; t = v \vee \exists t' : t \to t'$

Durch Induktion über eine Typherleitung für t läßt sich diese Eigenschaft recht einfach zeigen.

Als Konsequenz ergibt sich, daß die Auswertung von typisierbaren Termen niemals aufgrund von Typfehlern ins Stocken geraten kann.

Der Typ eines Terms bleibt unter Reduktion mit $\to$ erhalten:

Typerhaltung	$\Gamma \vdash t : T \wedge t \to t' \;\; \Rightarrow \;\; \Gamma \vdash t' : T$

Läßt man den Fixpunktoperator `fix` weg, hat der dabei entstehende Kalkül $\lambda_{\rightarrow}$ eine sehr interessante Eigenschaft, die für uns später noch von Bedeutung sein wird.

Im ungetypten λ-Kalkül können Fixpunktkombinatoren als Term dargestellt werden; z.B.[5]

$$\mathtt{Y} := \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Im getypten Kalkül $\lambda_{\rightarrow}$ ist `Y` kein gültiger Term, da ihm kein Typ zugeordnet werden kann. Grund ist mitunter die Untypisierbarkeit der Selbstapplikation $x\ x$. Ohne Fixpunktkombinator kann man keine Rekursion ausdrücken und auch keine partiellen Funktionen beschreiben. Es ergibt sich folgende Eigenschaft:

Jeder typisierbare Term t besitzt in $\lambda_{\rightarrow}$ eine Normalform, d.h.:

Normalisierung (ohne `fix`)	$\Gamma \vdash t : T \Rightarrow \exists v : t \xrightarrow{*} v$

Mit anderen Worten ist garantiert, daß die Berechnung eines wohlgetypten Terms terminiert.

Beweise für diese Eigenschaften finden sich in diversen Büchern und Arbeiten über den λ-Kalkül (z.B. [142, 13]).

2.4 Exkurs: Partielle Auswertung

Die Semantik eines Programmes kann man sich abstrakt gesehen auch als Funktion vorstellen, die auf einem Vektor von Eingabewerten operiert. Was ist nun, wenn nicht alle Werte dieses Vektors zur Verfügung stehen, sondern nur ein Teil? Dann kann man ein Programm partiell auswerten.

Die partiell vorhandenen Eingabewerte lassen viele Variablen in einem Programm zu Konstanten werden, wodurch im wesentlichen zwei Optimierungen möglich werden:

- *Constant folding*: ein Ausdruck oder Teilausdruck besteht nur noch aus Konstanten und kann vollständig ausgewertet werden.
- *Function unfolding*: Hier wird ein Funktionsaufruf textuell durch den Rumpf der aufgerufenen Funktion ersetzt. Gilt zum Beispiel $f(x) = x + 3$, dann kann der Ausdruck $f(4) + f(2)$ zu $4 + 3 + 2 + 3$ entfaltet werden.

Betrachten wir ein einfaches Beispiel. Gegeben sei die Funktion

$$f(x, y) = \mathtt{if}\ y \neq 0\ \mathtt{then}\ f(x + 1, y - 1) + y\ \mathtt{else}\ 2 * x$$

Angenommen, wir wüßten, daß der Parameter y den Wert 2 hat. Dann kann man obiges Programm wie folgt spezialisieren:

Zunächst setzen wir die bekannten Werte in die Funktionsgleichung ein:

$$f(x, 2) \quad = \quad \mathtt{if}\ 2 \neq 0\ \mathtt{then}\ f(x + 1, 2 - 1) + 2\ \mathtt{else}\ 2 * x$$

[5] Wir geben hier den einfacheren Fixpunktkombinator für die *call by name* Auswertungsstrategie an.

Offensichtlich trägt nur der `then`-Zweig zum Ergebnis bei und den Ausdruck $2-1$ können wir direkt zu 1 auswerten (*constant folding*); es ergibt sich:

$$f(x,2) = f(x+1,1)+2 \tag{2.1}$$

Wir fahren mit der Auswertung von $f(x+1,1)$ fort und erhalten durch Einsetzen:

$$f(x+1,1) \quad = \quad \texttt{if } 1 \neq 0 \texttt{ then } f(x+1+1,1-1)+1 \texttt{ else } 2*x$$

Auch in diesem Fall wird das Ergebnis ausschließlich vom `then`-Zweig getragen und der Ausdruck $1-1$ kann sofort zu 0 reduziert werden:

$$f(x+1,1) = f(x+2,0)+1$$

Einsetzen der rechten Seite in Gleichung 2.1 (*function folding*) führt zu:

$$f(x,2) = (f(x+2,0)+1)+2$$

Der Ausdruck $f(x+2,0)$ liefert offenbar den Wert $2*x$, so daß man zur Gleichung

$$f(x,2) = (2*x+1)+2$$

übergehen kann (*function folding*). Da f jetzt nicht mehr von y abhängt setzen wir

$$f^{res}(x) = (2*x+1)+2$$

Das neue Programm f^{res} nennt man **Residuum**.

Grob betrachtet kann man die Arbeit eines partiellen Auswerters so beschreiben, daß die Variablen eines Programmes in die zwei Klassen „statisch" und „dynamisch" eingeteilt werden – je nachdem, ob ihr Wert bekannt oder unbekannt ist. Ausgehend von dieser Einteilung wird versucht, Ausdrücke so weit wie möglich zu reduzieren. Leider ist dies nicht immer so einfach möglich, wie in unserem obigen Beispiel.

Angenommen es wäre nicht der Wert von y, sondern der von x bekannt (z.B. $x=5$). Dann würde unsere Strategie zu einer unendlichen Entfaltung von f führen:

$$\begin{aligned} f(5,y) &= \texttt{if } y=0 \texttt{ then } f(6,y-1) \texttt{ else } 10 \\ &= \texttt{if } y=0 \texttt{ then } (\texttt{if } y-1=0 \texttt{ then } f(7,y-2) \texttt{ else } 12) \texttt{ else } 10 \\ &= \cdots \end{aligned}$$

Die Ursache für dieses Dilemma liegt in der Abbruchbedingung, die allein vom dynamischen und damit unbekannten Wert von y abhängt.

Das Problem ließe sich dadurch aus dem Weg räumen, daß man bei der Spezialisierung der Funktion f das konkrete Argument, das an x gebunden wird, grundsätzlich als dynamisch ansieht - diese Vorgehensweise nennt man **Generalisierung**.

Abhängig vom Zeitpunkt, zu dem entschieden wird, ob eine Generalisierung notwendig ist, spricht man von *offline*- oder *online partial evaluation*.

Bei der *online partial evaluation* wird während der Spezialisierung generalisiert. Um unendliche Expansion zu vermeiden, könnte man bei der Spezialisierung die Aufrufspur (*calltrace*) mitschreiben, um dann vor Funktionsreduktionen zu prüfen, ob man sich im Kreis dreht.

Ein *offline partial evaluator* verläuft in zwei Phasen. In der ersten Phase (*binding time analysis* - oder kurz *BTA* genannt) wird das Programm mit Annotationen versehen. Jeder Parameter erhält eines der Attribute *static* oder *dynamic*.

Aufrufe an benutzerdefinierte Funktionen werden mit *unfold* markiert, wenn sie an Ort und Stelle expandiert werden sollen (*function unfolding*), oder mit *residualize*, wenn ein Residuum zu generieren ist (dabei kann eine Spezialversion der jeweiligen Funktion entstehen, so wie f^{res} in unserem Beispiel). Schließlich werden Anwendungen von eingebauten Funktionen mit *reduce* markiert, falls sie direkt ausgewertet werden können, und mit *specialize*, falls eine spezielle Version des Ausdrucks generiert werden soll.

Die erste Phase der *offline* Methode ist unabhängig von den tatsächlichen statischen Werten. Es genügt zu wissen, *welche* Werte zur Spezialisierungszeit bekannt sein werden - das bringt in unserem Fall einige Vorteile gegenüber dem *online* Verfahren. In der zweiten Phase der *offline* Methode wird aus dem annotierten Programm und konkreten statischen Werten ein spezialisiertes Programm generiert.

Wir wollen den Begriff *partial evaluator* festhalten, indem wir ihn formal definieren:

Definition 2.17 (*partial evaluator*)**.** Gegeben sei eine beliebige Sprache **S**. Ein ***partial evaluator*** ist eine **S**-Programm *spec*, das ein Programm p der Sprache **S** und einen Vektor von statischen Eingaben v_s übernimmt und daraus ein neues **S**-Programm p^{res} generiert:

$$[\![spec]\!]_{\mathbf{S}}\ p\ v_s = p^{res}$$

Dabei wird vorausgesetzt, daß für jedes **S**-Programm p und jede dynamische Eingabe v_d gilt:

$$[\![p]\!]_{\mathbf{S}}\ v_s\ v_d = [\![p^{res}]\!]\ v_d$$

Das Programm p^{res} heißt **Residuum**. ◊

2.5 Terme, die von Typen abhängen: Polymorpher λ-Kalkül - System F

Angenommen, wir wollen eine Funktion zur Verfügung stellen, die eine Funktion vom Typ $T \to T$ zweimal auf ein Argument anwendet. Dann können wird diese in $\lambda^{\texttt{fix}}$ nicht allgemein formulieren, sondern müssen für alle denkbaren Typen eine spezielle Funktion vereinbaren:

$$\texttt{twice}_{\texttt{int}} = \lambda f : \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.f\ (f\ x)$$
$$\texttt{twice}_{\texttt{bool}} = \lambda f : \texttt{bool} \to \texttt{bool}.\lambda x : \texttt{bool}.f\ (f\ x)$$

Der Funktionsrumpf ist jeweils identisch. Beide Funktionen in einem Programm zu haben würde gegen das **Abstraktionsprinzip** verstoßen, wonach jede signifikante Funktionalität eines Programmes durch genau ein Stück Quelltext realisiert wird. Aus softwaretechnischer Sicht ist

man natürlich daran interessiert, ähnliche Programmfragmente zu einem zusammenzufassen, indem man die verschiedenen Teile parametrisierbar macht.

Der polymorphe λ-Kalkül (oft System **F** genannt) leistet genau das für den Fall von `twice`, indem er erlaubt, Terme über Typen zu parametrisieren. Eine allgemeingültige Funktion `twice` hätte in System **F** z.B. folgende Gestalt:

$$\texttt{twice} = \Lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f\ (f\ x)$$

Zwei Konstrukte sind im Vergleich zu $\lambda^{\texttt{fix}}$ neu: Typvariablen, für die wir kleine griechische Buchstaben verwenden, und Typ-Abstraktionen (Λ-Abstraktionen). Funktionen, die mit mehreren Typen kompatibel sind, nennt man **polymorph**.

Eine Funktion zur Mehrfachanwendung von integer-Funktionen kann durch Term-Typ-Applikation, der dritten neuen syntaktischen Form in System **F**, aus der polymorphen Funktion `twice` gewonnen werden:

$$\texttt{twice}[\texttt{int}] = \lambda f : \texttt{int} \times \texttt{int}.\lambda x : \texttt{int}.f\ (f\ x)$$

Das Ergebnis einer Typapplikation $(\Lambda\alpha.t)[T]$ nennt man auch **Instanz** der polymorphen Funktion $\Lambda\alpha.t$.

Polymorphen Funktionen muß natürlich ein Typ zugeordnet werden können. Der Typ von `twice` hängt vom Typ ab, auf den wir diese Funktion anwenden. Um diese Abhängigkeit auszudrücken schreiben wir den Typ von `twice` als $\forall\alpha.\alpha \to \alpha$, wobei der Allquantor unterstreicht, daß wir `twice` auf beliebige Typen anwenden können.

2.5.1 Syntax

Zur Beschreibung der Syntax, des Typsystems und der Semantik von System **F** geben wir jeweils nur die Erweiterungen gegenüber $\lambda^{\texttt{fix}}$ an (siehe Abbildung 2.5).

In System **F** unterscheiden wir zwischen monomorphen Typen (oder **Monotypes**) und polymorphen Typen (**Polytypes**). Letztgenannte dürfen Allquantoren enthalten, während Monotypes allein aus Grundtypen, Typvariablen und dem Typkonstruktor $\to$ zur Bildung von Funktionstypen aufgebaut sind. Term-Typ-Applikationen und λ-Abstraktionen haben wir auf Monotypes beschränkt – eine Begründung für diese Entscheidung werden wir in Kürze nachreichen.

2.5.2 Typsystem

Neben Typzuweisungen legen wir in der Typisierungsumgebung Γ jetzt auch Typvariablen ab und schreiben $\Gamma \vdash \alpha$, wenn α in der Umgebung Γ enthalten ist. Abbildung 2.6 zeigt die Typisierungsregeln für die Term-Typ-Applikation und die Typ-Abstraktion.

Da wir nur an Sprachen interessiert sind, bei denen Typen zur Laufzeit keine Rolle spielen, schränken wir den Fixpunktkombinator auf monomorphe Funktionen ein.

Das heißt natürlich nicht, daß eine polymorphe Funktion nicht rekursiv sein darf, wie folgender Pseudocode zur Berechnung der Länge einer polymorphen Liste zeigt:

Syntax — System **F**

$M ::=$		**Monotypes**
	`bool`	Wahrheitswerte
	`int`	Ganze Zahlen
	$M \to M$	Funktionstyp
	α	Typvariable
$T ::=$		**Polytypes**
	M	Monomorpher Typ
	$T \to T$	Polymorpher Funktionstyp
	$\forall\alpha.T$	Polymorpher Typ
$t ::=$		**Terme**
	$\ldots$	
	$\lambda x : M.t$	λ-Abstraktion (Eingeschränkt auf Monotypes)
	$\Lambda\alpha.t$	Λ-Abstraktion
	$t[M]$	Term-Typ-Applikation
$v ::=$		**Werte**
	$\ldots$	
	$\Lambda\alpha.t$	Typ-Abstraktion

Abbildung 2.5: Syntax von System **F** (Erweiterung von Abbildung 2.2).

$\Gamma \vdash t : T$ — System **F**

$$\frac{\Gamma, \alpha \vdash t : T}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.T}\ \textbf{(T-TAbs)} \qquad \frac{\Gamma \vdash t : \forall\alpha.T}{\Gamma \vdash t[T_1] : T[T_1/\alpha]}\ \textbf{(T-TApp)}$$

$$\frac{\Gamma \vdash t : M \to M}{\Gamma \vdash \text{fix t} : M}\ \textbf{(T-Fix)}$$

Abbildung 2.6: Typisierung von Term-Typ-Applikation und Λ-Abstraktion.

$$\texttt{length} := \Lambda\alpha.\texttt{fix}\ \lambda len : \texttt{List}(\alpha) \to \texttt{int}.\lambda l : \texttt{List}(\alpha).\ \texttt{if}\ (isNil[\texttt{List}(\alpha)]\ l)\ \texttt{then}\ 0\ \texttt{else}\ 1 + len(tail[\texttt{List}(\alpha)]\ l)$$

Die Einschränkung bewirkt vielmehr den Ausschluß von polymorpher Rekursion.

2.5.3 Operationelle Semantik

Terme können in Abhängigkeit von Typen formuliert werden. Sofern ein Term keiner Λ-Abstraktion entspricht, wird er reduziert (Regel **E-TApp**). Wird eine Λ-Abstraktion auf einen Typ angewendet, so ergibt sich der resultierende Wert genau wie bei der β-Reduktion durch Substitution der gebundenen Variable α durch das Argument T im Term t (siehe Abbildung 2.7).

$\boxed{t_1 \to t_2}$ System **F**

$$\frac{t \to t'}{t[T] \to t'[T]}\ \textbf{(E-TApp)} \quad (\Lambda\alpha.t)[T] \to t[T/\alpha]\ \textbf{(E-TAppAbs)}$$

Abbildung 2.7: Operationelle Semantik der Term-Typ-Applikation.

2.5.4 Prädikative und imprädikative Version von System F

In der Literatur findet man unterschiedliche Varianten von System **F**. Am häufigsten wird eine **imprädikative** Version vorgestellt, bei der sich Typ- und Termvariablen auch über polymorphe Typen erstrecken dürfen. Im Typ $T = \forall\alpha.\alpha \to \alpha$ ist α Platzhalter für alle gültigen Typen – einschließlich dem Typ T selbst.

Interpretiert man Typen als Beschreibungen für Mengen, führt diese Sichtweise jedoch leicht zu Widersprüchen, weil das zu definierende Objekt Teil der Menge ist, aus der es konstruiert wird. Nehmen wir z.B. die polymorphe Identitätsfunktion $\texttt{id} := \Lambda\alpha.\lambda x : \alpha.x$. Diese hat den Typ $\forall\alpha.\alpha \to \alpha$ und kann im imprädikativen Fall auch auf sich selbst angewendet werden:

$$\texttt{id}[\forall\alpha.\alpha \to \alpha]\ \texttt{id}$$

Wenn wir uns nun den Definitionsbereich von `id` vorstellen, so muß dieser die Funktion `id` selbst enthalten!

Um diesen Widerspruch zu vermeiden, schränkt man Typvariablen auf monomorphe Typen ein. Man spricht dann von einem **prädikativen** Kalkül. Die Beschränkung kann man bereits auf der syntaktischen Ebene erreichen. So ist in dem hier vorgestellten Kalkül $\texttt{id}[\forall\alpha.\alpha \to \alpha]\ \texttt{id}$ kein gültiger Term, da die Typapplikation nur auf monomorphen Typen definiert ist.

Schränkt man, so wie wir, auch Termvariablen in λ-Abstraktionen auf monomorphe Objekte ein, erhält man einem **pränex-prädikativen** Kalkül. Der Vorteil liegt in bekanntermaßen entscheidbaren Typrekonstruktionsverfahren und einer für unsere Zwecke besonders wichtigen Eigenschaft, auf die wir aber erst in Abschnitt 3.3 eingehen werden.

Der Nachteil des pränex-prädikativen Kalküls liegt im Verlust an Ausdruckskraft, da Allquantoren nur noch auf der obersten Ebene eines Typausdrucks erscheinen können. In Abschnitt 2.7.1 werden wir zeigen, wie man diese Einschränkung entschärfen kann.

2.5.5 Eigenschaften von System F

System **F** hat hinsichtlich *Fortschritt*, *Typerhaltung* und *Normalisierung* (ohne `fix`) dieselben Eigenschaften wie der $\lambda^{\texttt{fix}}$-Kalkül.

Auch in System **F** kann für den Fixpunktkombinator kein Typ abgeleitet werden[6] und muß explizit in die Sprache aufgenommen werden, um partielle Funktionen realisieren zu können. Erstaunlicherweise ist System **F** aber auch ohne Fixpunktkombinator sehr ausdrucksstark – es läßt sich z.B. die bekanntermaßen nicht loop-berechenbare Ackermann Funktion codieren (siehe [147]).

Beweise zu den Eigenschaften von System **F** finden sich zum Beispiel in [55, 142, 13].

Für die überwiegende Zahl von Programmiersprachen spielen Typen nur zur Übersetzungszeit eine Rolle – das gilt auch für System **F**, denn wie man sich leicht überzeugt, ist das Laufzeitverhalten von System **F** unabhängig von der Typinformation (zumindest in dem Sinne, daß keine Programmverzweigung aufgrund von Typinformation erfolgt).

In der Tat kann man einen typsicheren System **F**-Term recht einfach in einen äquivalenten Term des ungetypten λ-Kalküls umformen (siehe z.B. [142, Seite 358]) – diese Umwandlung nennt man **Typelimination** (engl.: *type erasure*).

2.6 Typen, die von Typen abhängen: System $\mathbf{F}_\omega$

Im System $\mathbf{F}_\omega$ können Typen abhängig von Typen formuliert werden; z.B. beschreibt $\Lambda\alpha.\alpha \to \alpha$ eine Funktion, die einen Typ als Argument erwartet und einen Funktionstyp als Ergebnis liefert.

Um syntaktisch korrekte, aber semantisch fragwürdige Operationen über Typen herauszufiltern, stellen wir ein Typsystem für Typen und Operatoren über Typen zur Verfügung. Den Typ eines Typs nennen wir Gattung (engl. **Kind**).

Allen gültigen monomorphen Typen ordnen wir die Gattung `*` zu. Zum Beispiel sind `int`, `int` $\to$ `int` und `int` $\to$ `bool` alle von der Gattung `*`. Eine Operation, die einen Typ als Argument übernimmt und daraus einen Typ generiert gehört zur Gattung $* \Rightarrow *$.

In System $\mathbf{F}_\omega$ gibt es neben dem Konzept der Term-Typ-Applikation zur Instanziierung eines Terms, das Konzept der Typ-Typ-Applikation zur Instanziierung eines Typs.

2.6.1 Syntax

Auch hier geben wir nur die Erweiterungen gegenüber System **F** an (siehe Abbildung 2.8).

[6] Die Selbstapplikation ist in imprädikativen Varianten von System **F** typisierbar:
`selfApp` $:= \lambda x : \forall\alpha.\alpha \to \alpha.x[\forall\alpha.\alpha \to \alpha]x$.

System $\mathbf{F}_\omega$

$K ::=$		**Kinds**
	$*$	Kind von gültigen Typen
	$K \Rightarrow K$	Kind eines Typoperators
$M ::=$		**Monomorphe Typen**
	$\ldots$	
	$\Lambda\alpha :: K.M$	Typoperator
	$T[M]$	Typ-Typ-Applikation
$T ::=$		**Typen**
	$\ldots$	
	$\forall\alpha :: K.T$	Polymorpher Typ (mit Kinding-Information)
	$\Lambda\alpha :: K.T$	Typoperator
$t ::=$		**Terme**
	$\ldots$	
	$\Lambda\alpha :: K.t$	Typ-Abstraktion (mit Kinding-Information)
$v ::=$		**Werte**
	$\Lambda\alpha :: K.t$	Typ-Abstraktionen (mit Kinding-Information)

Abbildung 2.8: Syntax von System $\mathbf{F}_\omega$(Erweiterung von 2.5).

2.6.2 Kinding und Typäquivalenz

Kinding

Neben Typzuweisungen speichern wir in Γ jetzt auch Kind-Zuweisungen, die wir als $T :: K$ notieren. Die Grundtypen `int` und `bool` sind vom Kind `*`, anderen Typen ordnen wir über folgendes Deduktionsystem eine Gattung zu:

$\boxed{\Gamma \vdash T :: K}$ System $\mathbf{F}_\omega$

$$\frac{T :: K \in \Gamma}{\Gamma \vdash T :: K}\ \textbf{(K-TVar)} \qquad \frac{\Gamma, \alpha :: K \vdash T :: K'}{\Gamma \vdash \Lambda\alpha :: K.T :: K \Rightarrow K'}\ \textbf{(K-Abs)}$$

$$\frac{\Gamma \vdash T_1 :: K \Rightarrow K' \quad \Gamma \vdash T_2 :: K}{\Gamma \vdash T_1[T_2] :: K'}\ \textbf{(K-App)} \qquad \frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *}\ \textbf{(K-Arrow)}$$

$$\frac{\Gamma, \alpha :: K \vdash T :: *}{\Gamma \vdash \forall\alpha :: K.T :: *}\ \textbf{(K-All)}$$

Typäquivalenz

Betrachten wir die Identitätsfunktion für Typen:

$$\mathtt{id}_\mathbb{T} = \Lambda\alpha.\alpha$$

Dann bezeichnen die folgenden Ausdrücke alle denselben Funktionstyp:

$$\begin{array}{lll} \mathtt{int} \to \mathtt{bool} & \mathtt{id}_\mathbb{T}[\mathtt{int}] \to \mathtt{bool} & \mathtt{int} \to \mathtt{id}_\mathbb{T}[\mathtt{bool}] \\ \mathtt{id}_\mathbb{T}[\mathtt{int}] \to \mathtt{id}_\mathbb{T}[\mathtt{bool}] & \mathtt{id}_\mathbb{T}[\mathtt{int} \to \mathtt{bool}] & \mathtt{id}_\mathbb{T}[\mathtt{id}_\mathbb{T}[\mathtt{int} \to \mathtt{bool}]] \end{array}$$

Um diese Sichtweise zu präzisieren, legen wir fest, wann genau wir zwei Typen T_1 und T_2 als äquivalent annehmen wollen (in Zeichen $T_1 \equiv T_2$):

$\boxed{\Gamma \vdash T_1 \equiv T_2}$ System $\mathbf{F}_\omega$

$$\Gamma \vdash T \equiv T \quad \textbf{(Q-Refl)} \qquad \frac{\Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash T_2 \equiv T_1} \quad \textbf{(Q-Symm)}$$

$$\frac{\Gamma \vdash T_1 \equiv T_2 \quad T_2 \equiv T_3}{\Gamma \vdash T_1 \equiv T_3} \quad \textbf{(Q-Trans)} \qquad \frac{\Gamma \vdash T_1 \equiv T_2 \;\; T_3 \equiv T_4}{\Gamma \vdash T_1 \to T_2 \equiv T_3 \to T_4} \quad \textbf{(Q-Arrow)}$$

$$\frac{\Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash \Lambda\alpha :: K.T_1 \equiv \Lambda\alpha :: K.T_2} \quad \textbf{(Q-Abs)} \qquad \frac{\Gamma \vdash T_1 \equiv T_2 \;\; T_3 \equiv T_4}{\Gamma \vdash T_1[T_2] \equiv T_3[T_4]} \quad \textbf{(Q-App)}$$

$$\frac{\Gamma \vdash T_1 \equiv T_2}{\Gamma \vdash \forall\alpha :: K.T_1 \equiv \forall\alpha :: K.T_2} \quad \textbf{(Q-All)} \qquad \Gamma \vdash (\Lambda\alpha :: K.T_1)[T_2] \equiv T_1[T_2/\alpha] \quad \textbf{(Q-AppAbs)}$$

Fokussieren wir uns für einen Moment auf die Regeln **Q-Refl**, **Q-App** und **Q-AppAbs** und stellen uns vor, in der Regel sei das Symbol $\equiv$ durch das Symbol $\Downarrow$ ersetzt. Dann entsprechen diese Regeln exakt den Regeln einer *big-step* Semantik für den einfachen λ-Kalkül.

Wenn wir die Äquivalenzrelation zwischen Typen als Reduktionssystem interpretieren, können wir in System $\mathbf{F}_\omega$ folglich auf zwei Ebenen rechnen – auf diese Idee kommen wir später zurück.

2.6.3 Typsystem

Im Typsystem zu System $\mathbf{F}_\omega$ müssen wir die Verbindung zu den Kinding-Regeln herstellen. Wir müssen darauf achten, daß Typen, die in Termen auftauchen, ein Kind zugeordnet werden kann. Nur dann handelt es sich ja um gültige Typen.

Darüberhinaus müssen wir einige Regeln so erweitern, daß die neu eingeführte Äquivalenz von Typen berücksichtigt wird. Exemplarisch geben wir nur die geänderte Regel `T-If` an:

$\boxed{\Gamma \vdash t : T}$ System $\mathbf{F}_\omega$

$$\frac{\Gamma \vdash M :: * \;\; \Gamma, x : M \vdash t : T'}{\Gamma \vdash \lambda x : M.t : M \to T'} \quad \textbf{(T-Abs)} \qquad \frac{\Gamma \vdash t : T_1 \;\; T_1 \equiv T_2 \;\; \Gamma \vdash T_2 :: *}{\Gamma \vdash t : T_2} \quad \textbf{(T-Equiv)}$$

$$\frac{\Gamma, X :: K \vdash t : T}{\Gamma \vdash \Lambda\alpha :: K.t : \forall\alpha :: K.T} \quad \textbf{(T-TAbs)} \qquad \frac{\Gamma \vdash t : \forall\alpha :: K.T \;\; \Gamma \vdash T_1 :: K}{\Gamma \vdash t[T_1] : T[\alpha/T_1]} \quad \textbf{(T-TApp)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_c : T_c \quad \Gamma \vdash t_t : T_t \quad \Gamma \vdash t_e : T_e \\ \Gamma \vdash T_c \equiv \texttt{bool} \quad \Gamma \vdash T_t \equiv T_e\end{array}}{\Gamma \vdash \texttt{if } t_c \texttt{ then } t_t \texttt{ else } t_e : T} \quad \textbf{(T-If)}$$

2.6.4 Eigenschaften von System $\mathbf{F}_\omega$

In System $\mathbf{F}_\omega$ kann man mit Typen rechnen und Typen werden durch Gattungen qualifiziert. Die Frage des Fortschritts, der Typverträglichkeit und der Normalisierung kann man sich daher auch für Typausdrücke stellen.

Gehen wir zunächst der Frage nach, ob Terme die Eigenschaften *Fortschritt* und *Typerhaltung* erfüllen und lassen die Frage der Normalisierung zunächst außen vor – ein Beweis hierzu findet sich zum Beispiel bei [142].

Man könnte zunächst versuchen, Fortschritt und Typerhaltung ähnlich wie für den $\lambda^{\mathtt{fix}}$-Kalkül und System **F** durch Induktion über Typherleitungen zu beweisen. Der induktive Beweis war in diesen Fällen möglich, da sich die Typisierbarkeit eines Terms allein aus der Typisierbarkeit seiner Unterterme ergibt – der Beweis verläuft also völlig *syntaxgesteuert*.

Ein syntaxgesteuerter Beweis ist für System $\mathbf{F}_\omega$ nicht so ohne weiteres möglich, da mit der Regel **T-Equiv** der Zusammenhang zwischen Syntax und Typherleitung verloren geht: Zu einem Term t wird ein anderer Typ gesucht.

Man löst dieses Problem durch einen Trick, der für uns besonders interessant ist, da er die Vorstellung im Typsystem zu rechnen besonders unterstreicht.

Die Idee ist, eine Reduktionsrelation $\rightsquigarrow$ zu entwickeln, deren reflexiver und transitiver Abschluß $\overset{*}{\longleftrightsquigarrow}$ genau der Äquivalenzrelation auf Typen entspricht - Abbildung 2.9 zeigt die Definition von $\rightsquigarrow$.

$T_1 \rightsquigarrow T_2$ — System $\mathbf{F}_\omega$

$$T \rightsquigarrow T \quad \textbf{(QR-Refl)}$$

$$\frac{T_1 \rightsquigarrow T_1' \quad T_2 \rightsquigarrow T_2'}{T_1 \to T_2 \rightsquigarrow T_1' \to T_2'} \ \textbf{(QR-Arrow)} \qquad \frac{T \rightsquigarrow T'}{\Lambda\alpha :: K.T \rightsquigarrow \Lambda\alpha :: K.T'} \ \textbf{(QR-Abs)}$$

$$\frac{T_1 \rightsquigarrow T_1 \quad T_2 \rightsquigarrow T_2'}{T_1[T_2] \rightsquigarrow T_1'[T_2']} \ \textbf{(QR-App)} \qquad \frac{T \rightsquigarrow T' \quad T_2 \rightsquigarrow T_2'}{\Lambda\alpha :: K.T[T_2] \rightsquigarrow T'[T_2'/\alpha]} \ \textbf{(QR-AppAbs)}$$

Abbildung 2.9: Zur Typäquivalenzrelation $\equiv$ gehörende Reduktionsrelation $\rightsquigarrow$ – Reduktionssemantik für Typen.

Die Regeln zu $\rightsquigarrow$ entsprechen im wesentlichen den Regeln einer *big-step* Semantik für einen getypten λ-Kalkül. Die Typregeln für diesen Kalkül entsprechen gerade den Regeln zum Kinding.

In diesem virtuellen Kalkül kann der Selbstapplikation $\Lambda\alpha :: K.\alpha[\alpha]$ kein Kind zugeordnet werden. Dies ist ein gutes Indiz dafür, daß man in diesem Kalkül keinen Fixpunktkombinator ausdrücken kann und folglich jeder typisierbare Term normalisierend ist.

In der Tat kann man zeigen, daß $\rightsquigarrow$ konfluent und streng normalisierend ist (siehe z.B. [118][142]).

Bei der Typüberprüfung von System $\mathbf{F}_\omega$ kann man jetzt auf die Berücksichtigung von äquivalenten Typen verzichten, wenn man vereinbart, alle Typen zunächst mit $\rightsquigarrow$ zu einer Normalform zu reduzieren. Die Äquivalenz zweier Typen ergibt sich dann aus der Gleichheit ihrer Normalformen.

Damit wird die Typherleitung wieder rein syntaxgesteuert und die Beweise zu *Fortschritt* und *Typerhaltung* lassen sich analog zu den entsprechenden Beweisen der anderen Calculi führen.

Auch in System $\mathbf{F}_\omega$ können sämtliche Typberechnungen zur Übersetzungszeit durchgeführt werden, da Typen auch hier keine gültigen Werte sind und es kein Sprachkonstrukt gibt, welches eine Aktion abhängig von einem Typ macht.

2.7 Erweiterung von System $\mathbf{F}_\omega$

2.7.1 `Let`-Ausdrücke

Als einfache Erweiterung stellen wir sogenannte `let`-Ausdrücke vor. Ein `let`-Ausdruck ist im Grunde genommen syntaktischer Zucker, um Teilausdrücke aus einem Ausdruck zu extrahieren. Zum Beispiel kann man

$$\lambda x : T.(x + x) * (x + x)$$

mit einem `let`-Ausdruck umschreiben zu:

$$\begin{aligned}\lambda x.&\texttt{let } z = x * x\\ &\texttt{in } z + z\end{aligned}$$

Syntax — System $\mathbf{F}_\omega$ + `let`

$$\begin{array}{lll} t ::= & & \textbf{Terme}\\ & \cdots & \\ & \texttt{let } x = t \texttt{ in } t & \text{Let-Ausdruck}\end{array}$$

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash t_r[t/x] : M}{\Gamma \vdash \texttt{let } x = t \texttt{ in } t_r : M} \quad \textbf{(T-Let)}$$

$t_1 \to t_2$

$$\frac{t \to t'}{\texttt{let } x = t \texttt{ in } t_r \to \texttt{let } x = t' \texttt{ in } t_r} \quad \textbf{(E-Let-Step)}$$

$$\texttt{let } x = v \texttt{ in } t_r \to t_r[v/x] \quad \textbf{(E-Let)}$$

Abbildung 2.10: Erweiterung von System $\mathbf{F}_\omega$ um `let`-Ausdrücke.

Die eigentliche Stärke von `let`-Ausdrücken verbirgt sich hinter der Typisierungsregel zu `let`, da sie uns ein wenig von der Flexibilität zurückgibt, die wir durch die Einschränkung auf einen pränex-prädikativen Kalkül verloren haben.

Im Unterschied zu λ-Abstraktionen darf die Variable x auch an einen polymorphen Term t gebunden werden. Den Typ von t_r bestimmen wir, indem wir die Variable x *vor* der Typberechnung durch den Term t ersetzen. War t polymorph, so können in t_r unterschiedliche Instanzen von t erzeugt werden; z.B.

$$\begin{aligned}&\texttt{let twice} = \Lambda\alpha :: *.\lambda f : \alpha \to \alpha.\lambda a : \alpha.f\ (f\ a)\\ &\quad f = \lambda x : \texttt{bool}.!x\\ &\quad g = \lambda x : \texttt{int}.x + 1\\ &\quad \texttt{in } (\texttt{twice}[\texttt{bool}]\ f\ \texttt{true}, \texttt{twice}[\texttt{int}]\ g\ 3)\end{aligned}$$

Das Ergebnis eines `let`-Ausdrucks muß ein monomorpher Wert sein.

Im Vergleich zu einer λ-Abstraktion mit polymorphem Argument, kommen wir bei der Typzuweisung zu einem `let`-Term ohne Allquantoren aus. Der Trick ist, daß die Typisierungsregel von `let` im Grunde genommen einen β-Reduktionsschritt vollzieht und durch die Forderung, daß der Term t_r monomorph sein muß, garantiert ist, daß die dafür notwendigen Typargumente auch zur Verfügung stehen.

Zur Auswertung des `let`-Ausdrucks $\texttt{let}\ x = t\ \texttt{in}\ t_r$, wird zunächst eine Normalform zu t bestimmt (wiederholtes Anwenden der Regel **E-Let-Step**). Anschließend werden alle Vorkommen der Variable x im Term t_r durch diesen Wert ersetzt.

Syntax sugar

Häufig möchte man mehrere Teilterme in einen `let`-Ausdruck auslagern, was zu tief verschachtelten Termen führen kann. In einer erweiterten konkreten Syntax fassen wir daher mehrere Bindungen zusammen und schreiben z.B. anstelle von

$$\begin{array}{l}\texttt{let}\ x_1 = t_1\\ \quad\texttt{in let}\ x_2 = t_2\\ \qquad\texttt{in let}\ x_3 = t_3\\ \qquad\quad\texttt{in}\ t\end{array}$$

auch

$$\begin{array}{ll}\texttt{let} & x_1 = t_1\\ & x_2 = t_2\\ & x_3 = t_3\\ & \texttt{in}\ t\end{array}$$

Zu beachten ist, daß sich an der Sichtbarkeit der Variablen nichts ändert. x_1 kann nach wie vor zur Definition von x_2 oder x_3 verwendet werden.

2.7.2 Konstruktortypen

Konstruktortypen können als eine Spezialform von Produkttypen (Tupeln) angesehen werden, bei denen jedes Tupel mit einem Kopfsymbol (einem Konstruktornamen) markiert wird.

Zur Bildung von Konstruktortypen setzen wir eine Signatur $\Sigma_{\mathcal{C}} = \langle \mathcal{C}, \sigma_{\mathcal{C}} \rangle$ voraus, wobei $\mathcal{C}$ eine endliche Menge von **Konstruktoren** ist.

In einer konkreten Programmiersprache würde sich $\Sigma_{\mathcal{C}}$ unmittelbar aus den Datentypdefinitionen eines Programmes und den eventuell vorgegebenen Konstruktoren zusammensetzen. Entsprechende Sprachkonstrukte ließen sich zwar leicht zu System $\mathbf{F}_\omega$ hinzunehmen, würden sich allerdings als hinderlich bei den weiteren Untersuchungen erweisen.

Wie schon angedeutet, kann man sich die Werte von Konstruktortypen als benamte n-Tupel vorstellen. Zum Beispiel sind Werte vom Typ $\texttt{C}(\texttt{int}, \texttt{bool})$ im Grunde genommen Paare aus integer- und booleschen Werten (also aus $\texttt{int} \times \texttt{bool}$), wobei wir dem Paar den Konstruktornamen `C` voranstellen (z.B. $\texttt{C}(3, \texttt{true})$).

Wir setzen Konstruktoren sowohl zur Konstruktion von Werten, als auch zur Konstruktion von Typen ein. Die Benutzung eines Konstruktors wird nicht eingeschränkt: Konstruktoren

System $\mathbf{F}_\omega$ + Kons

$T ::=$		**Typen**
	$\cdots$	
	$\mathtt{C}(T, \cdots, T)$	Mehrstelliger Konstruktor
	$\mathtt{C}$	Konstruktor-Konstante
$t ::=$		**Terme**
	$\cdots$	
	$\mathtt{C}(t, \cdots, t)$	Konstruktor
	$\mathtt{C}$	Konstruktor-Konstante
	$t.i$	Projektion ($i \in \mathbb{N}$)
$v ::=$		**Werte**
	$\cdots$	
	$\mathtt{C}(v, \cdots, v)$	Konstruktor
	$\mathtt{C}$	Konstruktor-Konstante

$\boxed{\Gamma \vdash T :: *}$

$$\frac{\mathtt{C} \in \mathcal{C} \quad \sigma_{\mathcal{C}}(\mathtt{C}) = 0}{\Gamma \vdash \mathtt{C} :: *} \textbf{ (K-Cons-0)}$$

$$\frac{C \in \mathcal{C} \quad \sigma_{\mathcal{C}}(\mathtt{C}) = 1 \quad \Gamma \vdash T :: *}{\Gamma \vdash \mathtt{C}(T) :: *} \textbf{ (K-Cons-App-1)}$$

$$\frac{C \in \mathcal{C} \quad \sigma_{\mathcal{C}}(\mathtt{C}) = n \quad \Gamma \vdash T_1 :: * \quad \cdots \quad \Gamma \vdash T_n :: *}{\Gamma \vdash \mathtt{C}(T_1, \cdots, T_n) :: *} \textbf{ (K-Cons-App-N)}$$

$$\frac{C \in \mathcal{C} \quad \sigma_{\mathcal{C}}(\mathtt{C}) = n \quad \Gamma \vdash T_1 \equiv S_1 \quad \cdots \quad \Gamma \vdash T_n \equiv S_n}{\Gamma \vdash \mathtt{C}(T_1, \cdots, T_n) \equiv \mathtt{C}(S_1, \cdots, S_n)} \textbf{ (Q-Cons)}$$

$\boxed{\Gamma \vdash t : T}$

$$\frac{\sigma_{\mathtt{C}} = 0 \quad \mathtt{C} \notin \{\mathtt{int}, \mathtt{bool}, \mathtt{void}\}}{\Gamma \vdash \mathtt{C} : \mathtt{C}} \textbf{ (T-Cons-0)} \qquad \frac{\sigma_{\mathtt{C}} = 1 \quad \Gamma \vdash t : T}{\Gamma \vdash \mathtt{C}(t) : \mathtt{C}(T)} \textbf{ (T-Cons-1)}$$

$$\frac{\sigma_{\mathtt{C}} = n \quad \Gamma \vdash t_1 : T_1 \quad \cdots \quad \Gamma \vdash t_n : T_n \quad \mathtt{C} \neq \rightarrow}{\Gamma \vdash \mathtt{C}(t_1, \cdots, t_n) : \mathtt{C}(T_1, \cdots, T_n)} \textbf{ (T-Cons-N)}$$

$$\frac{\sigma_{\mathtt{C}} = n \quad \Gamma \vdash t : \mathtt{C}(T_1, \cdots, T_n) \quad 0 \leq i \leq n-1}{\Gamma \vdash t.i : T_{i+1}} \textbf{ (T-ConsProj)}$$

Abbildung 2.11: Syntax, Kinding und Typsystem für Konstruktortypen.

akzeptieren als Typkonstruktor beliebige Typen und als Wertkonstruktor beliebige Werte. Lediglich die Stelligkeit eines Konstruktors ist verbindlich.

$\texttt{C}(\texttt{double}, \texttt{bool})$, $\texttt{C}(\texttt{int}, \texttt{bool})$ und $\texttt{D}(\texttt{int}, \texttt{bool})$ sind gültige, verschiedene Typen. $\texttt{C}(1, 2)$ und $\texttt{C}(\texttt{FALSE}, 3)$ sind gültige Werte von unterschiedlichem Typ.

Der Vorteil von Konstruktoren gegenüber einfachen Tupeln liegt darin, daß wir den Verwendungszweck eines Wertes mit Hilfe des Konstruktornamens einschränken können. Zum Beispiel könnten wir vereinbaren, daß im Typ $\texttt{Vec}_{cm}(\texttt{int}, \texttt{int})$ in beiden Komponenten Größenangaben in Zentimetern abgelegt sind; im Typ $\texttt{Vec}_{m}(\texttt{int}, \texttt{int})$ hingegen in Metern. Die Größenangabe ist Bestandteil der Typinformation, so daß man den Definitionsbereich von Funktionen derart einschränken kann, daß sie nur mit einer bestimmten Vektorsorte verträglich ist.

Wir setzen nachfolgend voraus, daß `int` und `bool` nullstellige Konstruktoren und $\rightarrow$ ein binärer Konstruktor ist, den wir - wie gewohnt - in infix-Notation verwenden. Allerdings lassen wir `int` und `bool` nicht als Werte zu, sondern verwenden hier die übliche Interpretation[7].

Als Besonderheit vereinbaren wir noch die Existenz eines Konstruktors `void`, dem wir eine leere Wertemenge zuordnen.

Die Reduktion eines Konstruktors beschreiben wir ausnahmsweise im Stil einer *big-step* Semantik. Ansonsten müssten wir für jede Konstruktorkomponente eine eigene Regel angeben, die alle vorangegangenen Komponenten als reduziert annimmt und den nächsten Term im Konstruktor reduziert.

$\boxed{t_1 \rightarrow t_2}$ System $\mathbf{F}_\omega$ + Kons

$$\frac{t_1 \Downarrow v_1 \quad t_n \Downarrow v_n}{\texttt{C}(t_1, \cdots, t_n) \rightarrow \texttt{C}(v_1, \cdots, v_n)} \textbf{ (E-Cons)} \qquad \frac{t \rightarrow t'}{t.i \rightarrow t'.i} \textbf{ (E-ConsProjStep)}$$

$$C(v_1, \cdots, v_n).i \rightarrow v_{i+1} \textbf{ (E-ConsProj)}$$

Konstruktortypen vs. rekursive Typen

In funktionalen Sprachen wie Haskell und ML gibt es keine Konstruktortypen – zumindest nicht so, wie wir sie hier eingeführt haben. Konstruktoren tauchen in diesen Sprachen niemals isoliert auf, sondern sind Bestandteil einer Definition für algebraische Datentypen. Betrachten wir als Beispiel einen polymorphen Listentyp:

```
IntList α = N | C(α, IntList α)
```

`IntList` ist ein polymorpher Summentyp, d.h. er beschreibt die Summe der Typen `N` und $\texttt{C}(\alpha, \texttt{IntList}\ \alpha)$ für einen bestimmten Typ α.

Offenbar ist `IntList` rekursiv definiert und es lassen sich folgende Typäquivalenzen ableiten:

$$\texttt{IntList}\ \alpha \equiv \texttt{N} \equiv \texttt{C}(\alpha, \texttt{N}) \equiv \texttt{C}(\alpha, \texttt{C}(\alpha, \texttt{N})) \cdots$$

Um nicht mit unendlich vielen Typen arbeiten zu müssen, beschränkt man sich auf den Typ $\texttt{IntList}\ \alpha$. Zum Beispiel wird dem Wert $\texttt{C}(1, \texttt{C}(2, \texttt{N}))$ nicht der Typ $\texttt{C}(\texttt{int}, \texttt{C}(\texttt{int}, \texttt{N}))$, sondern

[7] Nach Aufnahme aller Grundtypsymbole in $\Sigma_\mathcal{C}$ müsste die Syntax von System $\mathbf{F}_\omega$ natürlich entsprechend angepaßt werden.

der Typ `IntList int` zugewiesen – Konstruktoren werden in Haskell und ML primär zur Konstruktion von Werten verwendet.

Die Definition eines algebraischen Datentyps schränkt die Benutzbarkeit eines Konstruktors auf bestimmte Fälle ein. Zum Beispiel ist $\texttt{C}(\texttt{true}, \texttt{C}(\texttt{3}, \texttt{N}))$ kein gültiger Wert, da es kein α gibt, so daß $C(\texttt{bool}, C(\texttt{int}, N))$ äquivalent zu `IntList` α wäre.

Werte, die mit verschiedenen Konstruktoren erzeugt wurden, als typgleich zu interpretieren bringt noch einen weiteren Vorteil. Betrachten wir hierzu die Haskell-Funktion `len`, die die Länge einer Liste berechnet:

```
len x = case x of
          N         -> 0
          Cons h t  -> 1 + (len t)
```

`len` hat den Typ $\forall\alpha.\texttt{List}\ \alpha \rightarrow \texttt{int}$, ist also eine polymorphe Funktion, die auf beliebige Instanzen von `List` angewendet werden kann. Mit der `case`-Anweisung kann Programmcode abhängig vom Konstruktor, mit dem ein Listenwert erstellt wurde, ausgeführt werden. Die `case`-Analyse ist ein Laufzeit-Mechanismus.

Wir hatten bereits angesprochen, daß die Übersetzung einer polymorphen Funktion durch Typelimination möglich ist – das gilt auch für `len`. Voraussetzung für diese Übersetzungsstrategie ist natürlich, daß der generierte Programmcode mit allen Instanzen des Listentyps kompatibel ist bzw. daß alle Instanzen eines Listentyps in ähnlicher Art und Weise im Speicher hinterlegt sind.

Typischerweise wird dies durch den Einsatz von Zeigern realisiert und eine Liste wird im Zielcode als Zeigerpaar dargestellt. Der Zugriff auf das Kopfelement einer Liste erfordert dann aber einen indirekten Speicherzugriff, was insbesondere in Hinblick auf moderne Cache-Architekturen zu hohen Laufzeitkosten führen kann.

Eine andere Übersetzungsstrategie wäre, verschiedene Instanzen eines Listentyps unterschiedlich im Speicher darzustellen. Folglich müßten dann aber auch spezielle `len`-Funktionen für jeden dieser Typen generiert werden.

Die erste Übersetzungsstrategie nennt man **homogene Übersetzung**, da einem Stück Quelltext genau ein Stück Zielcode zugeordnet ist. Die zweite Strategie nennt man **heterogene Übersetzung** (siehe z.B. [128, 127]), sie resultiert in größerem, meist aber effizienterem Programmcode.

Vergleichen wir nun mit Konstruktortypen. Für sich genommen geben einem diese völlige Freiheit bei der Instanziieriung eines Typs aus einem Konstruktor. So kann man z.B. mit den Konstruktoren `C` und `N` Listen konstruieren, in denen der Elementtyp variiert (Polytyplisten)[8].

Betrachten wir die folgende Funktion zur Konstruktion solcher Listen:

$$\texttt{cons} = \Lambda\alpha :: *.\Lambda\beta :: *.\lambda x : \alpha.\lambda y : \beta.\texttt{C}(x, y)$$

Offenbar ist $\texttt{cons}[\texttt{int}][\texttt{C}(\texttt{bool}, \texttt{N})]\ \texttt{3}\ (\texttt{cons}[\texttt{bool}][\texttt{N}]\ \texttt{true}\ \texttt{N})$ ein wohlgetypter Ausdruck, der sich zum Wert $\texttt{C}(\texttt{3}, \texttt{C}(\texttt{true}, \texttt{N}))$ vom Typ $\texttt{C}(\texttt{int}, \texttt{C}(\texttt{bool}, \texttt{N}))$ reduzieren läßt.

[8] Im ersten Moment mag man einwenden, daß es sich hier weniger um Listen, als vielmehr um Tupel handelt. Wir werden später sehen, daß man mit dieser Konstruktion alle listentypischen Operationen (z.B. Konkatenation) realisieren kann, so daß der Programmierer Polytyplisten wirklich als Listen empfindet.

Typ und Wert korrespondieren hinsichtlich ihrer Struktur, so daß es unmöglich ist, in System $\mathbf{F}_\omega$ eine Funktion anzugeben, die die Länge einer Polytypliste berechnet. Das Problem liegt darin, daß wir in System $\mathbf{F}_\omega$ keine Möglichkeit haben, den Typ einer Funktion derart einzuschränken, daß sie nur gültige Polytyplisten akzeptiert.

Polymorphe Funktionen, bei denen der Elementtyp einer Liste relevant ist (zum Beispiel die Suche nach einem bestimmten Listenelement) lassen sich für Polytyplisten in System $\mathbf{F}_\omega$ ebenfalls nicht formulieren, da der Vergleich zweier Elemente typabhängig ist.

Um beide Probleme zu lösen, brauchen wir Sprachmittel, mit denen wir Typen und Terme in Abhängigkeit von der Struktur eines Konstruktortyps definieren können. Entsprechende Erweiterungen werden wir im nächsten Kapitel vorstellen.

Syntax sugar

Wie wir gesehen haben, kann man mit Konstruktortypen Polytyplisten konstruieren. Solche Listen werden wir in Zukunft öfter einsetzen und wollen daher eine suggestivere Notation für diese vereinbaren.

Eine n-elementige Liste $\mathtt{C}(a_1, \mathtt{C}(a_2, \cdots, \mathtt{N}))$ notieren wir in der Form

$$\{\!|a_1 : a_2 : \cdots : a_n|\!\}$$

Der Wert `N` wird in dieser Notation weggelassen. Er symbolisiert die leere Liste, die wir durch $\{\!|\,|\!\}$ beschreiben.

Analog verfahren wir für den Typ einer Liste und schreiben z.B. $\{\!|\mathtt{int} : \mathtt{bool}|\!\}$ für eine Liste, die einen integer- und ein booleschen Wert speichert.

Ferner setzen wir einen binären Konstruktor $\times$ voraus, den wir wie $\rightarrow$ in infix-Notation verwenden und der zur Beschreibung von Tupeltypen dient, bei denen uns der Konstruktorname nicht wichtig ist. Die Werte eines Tupeltyps schreiben wir wie üblich als (t_1, t_2) und eher selten als $\times(t_1, t_2)$.

$\times$ werden wir gelegentlich aber auch zur Bildung von namenlosen n-Tupeln einsetzen (z.B. $\mathtt{bool} \times \mathtt{int} \times \mathtt{int}$)[9]. Die Werte eines n-Tupels werden wir dann ebenfalls in der gebräuchlichen Notation schreiben.

2.7.3 Referenzen, Zuweisungen und Sequenzen

Die Calculi, die wir bis jetzt vorgestellt haben, sind alle frei von Seiteneffekten. Das Ergebnis einer Berechnung ist ausschließlich von den Funktionsargumenten abhängig und die Reihenfolge, in der Argumente ausgewertet werden, hat keinen Einfluß auf das Gesamtergebnis.

Will man einen globalen Status implementieren, so muß man diesen als Argument durch sämtliche Berechnungen hindurchschleifen. Das kostet in der Praxis natürlich Zeit und Speicherplatz, da der globale Zustandsraum mit jedem Funktionsaufruf erneut auf den Maschinenstack kopiert werden muß.

Zwar läßt sich das Problem auf Seiten der Implementierung dadurch entschärfen, daß man nicht den Zustandsraum selbst, sondern einen Zeiger auf diesen übergibt. Aber spätestens

[9] Hier entspricht $\times$ natürlich nicht mehr dem zweistelligen Konstruktor zur Bildung von Tupeln!

dann, wenn der Zustand manipuliert werden soll, muß eine Kopie angelegt werden, um eventuell existierende Referenzen auf den alten Zustand nicht zu beeinflussen.

Effizienter ist die Bereitstellung eines speziellen Freispeichers (manchmal auch *heap* genannt), in dem Objekte von beliebigem Typ hinterlegt werden können.

Abstrakt gesehen ist ein Freispeicher eine partielle Abbildung von einer Menge von Speicheradressen in die Menge der System $\mathbf{F}_\omega$-Werte.

Speicheradressen notieren wir als $\texttt{ref}(N)$, wobei N eine beliebiger, ganzzahliger Wert ist. Um einen Term t vom Typ T auf dem Freispeicher zu verschieben, verwendet man das Kommando $\texttt{new}(t)$. Im Ergebnis erhält man einen Wert vom Typ $\texttt{Ref}(T)$, also einen Zeiger auf eine Speicherzelle, in der ein Wert vom Typ T abgelegt ist.

Um auf den Wert einer Speicherzelle zuzugreifen, muß man den entsprechenden Zeiger mit dem Kommando `deref` dereferenzieren.

Einmal mit `new` angeforderte Speicherzellen können durch den Zuweisungsoperator verändert werden. Die Zuweisung $\texttt{ref}(N) := t$ weist der Speicherstelle $\texttt{ref}(N)$ den neuen Wert t zu. Voraussetzung ist, daß die Speicherzelle $\texttt{ref}(N)$ mit dem Typ von t kompatibel ist.

Das Ergebnis einer Zuweisung ist immer die Konstante `unit` vom Typ `Unit`. Eine Funktion, die integer-Speicherzellen mit Null initialisiert ($\lambda x : \texttt{Ref}(\texttt{int}).x := 0$) hat folglich den Typ $\texttt{Ref}(\texttt{int}) \to \texttt{Unit}$.

Durch Einsatz des Sequenzoperators `;` lassen sich mehrere Zuweisungen hintereinanderreihen. Zum Ergebnis trägt aber nur der letzte Term der Sequenz bei. Hier als Beispiel eine Funktion, die den Inhalt der Speicherzellen x und y vertauscht und die Summe der in den Speicherzellen gespeicherten Werte als Ergebnis zurückliefert:

$$\begin{aligned}\texttt{chgAdd} := \lambda x : \texttt{Ref}(\texttt{int}).\lambda y : \texttt{Ref}(\texttt{int}).x.\\ \texttt{let } h := \texttt{new}(\texttt{deref}(x))\\ \texttt{in } h := \texttt{deref}(x); x := \texttt{deref}(y); y := \texttt{deref}(h); \texttt{deref}(x) + \texttt{deref}(y)\end{aligned}$$

Die Funktion `chgAdd` hat den Typ $\texttt{Ref}(\texttt{int}) \to \texttt{Ref}(\texttt{int}) \to \texttt{int}$.

Abbildung 2.12 zeigt syntaktische Erweiterungen, sowie Typsystem und Semantik zu den neuen imperativen Merkmalen.

Um zu verhindern, daß der Programmierer willkürlich auf den Freispeicher zugreifen kann, geben wir keine Typregel für Zeiger an. Der Term $\texttt{ref}(0)$ ist syntaktisch zwar korrekt, wird vom Typsystem jedoch abgelehnt. Referenzen können ausschließlich durch Einsatz von `new` erzeugt werden.

Bei den Semantikregeln müssen wir den global sichtbaren Freispeicher berücksichtigen, für den wir die Metavariable μ verwenden. Die Reduktionsrelation ist vierstellig geworden: $t|\mu \to t'|\mu'$ bedeutet, daß der Term t in der Umgebung des Freispeichers μ zu t' reduziert werden kann, wobei als Seiteneffekt der Freispeicher μ' entsteht.

Soll ein neuer Wert auf den Freispeicher verschoben werden, so wird zunächst der an `new` übergebene Term reduziert. Diese Reduktion kann als Seiteneffekt die Änderung des Freispeichers von μ zu μ' hervorrufen. Wird ein Wert an `new` übergeben[10], so nehmen wir eine beliebige Adresse x, die noch nicht im Definitionsbereich $\mathit{domain}(\mu)$ enthalten ist, und ergänzen die partielle Abbildung μ um die Zuordnung $[x \mapsto v]$.

[10] $\texttt{ref}(x)$ zählt ja zu den Werten.

Syntax — System $\mathbf{F}_\omega$ + Imp

$t ::=$		**Terme**
	$\ldots$	
	$\texttt{ref}(n)$	Referenz ($n \in \mathbb{N}$)
	$\texttt{new}(t)$	new-Operator
	$\texttt{deref}(t)$	dereferenzier-Operator
	$\texttt{unit}$	unit-Konstante
	$t := t$	Zuweisung
	$t; t$	Sequenz
$v ::=$		**Werte**
	$\texttt{ref}(n)$	Referenz ($n \in \mathbb{N}$)
$T ::=$		**Typen**
	$\ldots$	
	$\texttt{Unit}$	Unit-Typ
	$\texttt{Ref}(T)$	Referenztyp

$\Gamma \vdash t : T$

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \quad \textbf{(T-Unit)} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash \texttt{new}(t) : \texttt{Ref}(T)} \quad \textbf{(T-new)}$$

$$\frac{\Gamma \vdash t : \texttt{Ref}(T)}{\Gamma \vdash \texttt{deref}(t) : T} \quad \textbf{(T-deref)} \qquad \frac{\Gamma \vdash t_1 : \texttt{Ref}(T) \;\; \Gamma \vdash t_2 : \texttt{Ref}(T)}{\Gamma \vdash t_1 := t_2 : \texttt{Unit}} \quad \textbf{(T-assign)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \;\; \Gamma \vdash T_2 : T}{\Gamma \vdash t_1; t_2 : T_2} \quad \textbf{(T-sequence)}$$

$t_1 \to t_2$

$$\frac{t_1|\mu \to t_2|\mu'}{\texttt{new}(t_1)|\mu \to \texttt{new}(t_2)|\mu'} \quad \textbf{(E-new-1)} \qquad \frac{x \notin \mathit{domain}(\mu)}{\texttt{new}(v)|\mu \to \texttt{ref}(x)|\mu[x \mapsto v]} \quad \textbf{(E-new-2)}$$

$$\frac{t_1|\mu \to t_2|\mu'}{\texttt{deref}(t_1)|\mu \to \texttt{deref}(t_2)|\mu'} \quad \textbf{(E-deref-1)} \qquad \texttt{deref}v|\mu \to \mu(v)|\mu \quad \textbf{(E-deref-2)}$$

$$\frac{t_2|\mu \to t_2'|\mu'}{t_1 := t_2|\mu \to t_1 := t_2'|\mu'} \quad \textbf{(E-assign-1)} \qquad \frac{t_1|\mu \to t_1'|\mu'}{t_1 := v|\mu \to t_1' := v|\mu'} \quad \textbf{(E-assign-2)}$$

$$\texttt{ref}(x) := v|\mu \to \texttt{unit}|\mu[x \mapsto v] \quad \textbf{(E-assign)}$$

$$\frac{t_1|\mu \to t_1'|\mu'}{t_1; t_2|\mu \to t_1'; t_2|\mu'} \quad \textbf{(E-sequence-1)} \qquad \frac{t_2|\mu \to t_2'|\mu'}{\texttt{unit}; t_2|\mu \rightsquigarrow t_2'|\mu'} \quad \textbf{(E-sequence-2)}$$

Abbildung 2.12: Erweiterung von System $\mathbf{F}_\omega$ um imperative Merkmale.

Auch beim Dereferenzieren wird zunächst der an `deref` übergebene Term reduziert. Steht hier ein Wert $\texttt{ref}(n)$, wird der zugehörige Wert aus dem Freispeicher ausgelesen und als Ergebnis zurückgegeben.

Die Reduktion einer Zuweisung startet mit der Reduktion der rechten Seite (**E-assign-1**). Konnte diese zu einem Wert v reduziert werden, wird mit der Reduktion der rechten Seite zu einer Referenz $\texttt{ref}(x)$ fortgefahren (**E-assign-2**). Abschließend wird der Freispeicher so geändert, daß an der Speicherstelle x der Wert v steht (**E-assign**).

Bei Sequenzen wird zuerst die linke Seite zum Wert `unit` reduziert. Dieser Wert wird jedoch ignoriert und stattdessen mit der Reduktion der rechten Seite fortgefahren.

Die Semantikregeln, die wir von System $\mathbf{F}_\omega$ übernommen haben, müssen natürlich so angepaßt werden, daß der Freispeicher berücksichtigt wird. Zum Beispiel muß die Regel **E-App-1** geändert werden zu:

$$\frac{t_1|\mu \rightarrow t_1'|\mu'}{t_1\ t_2|\mu \rightarrow t_1'|\mu'} \quad \textbf{(E-App-1)}$$

Referenzen und durch Zuweisung änderbare Speicherstellen machen das Beweisen von Programmeigenschaften sehr aufwendig. Zum Beispiel würde man auf den ersten Blick vermuten, daß die Funktion

$$\lambda x : \texttt{Ref}(\texttt{int}).\lambda y : \texttt{Ref}(\texttt{int}).x := 2; y := 3; \texttt{deref}(x)$$

grundsätzlich 2 als Ergebnis liefert. Das ist jedoch nicht der Fall, wenn sich die Referenzen x und y auf ein und dieselbe Speicherstelle beziehen (x und y also **Aliase** für eine Speicherzelle sind) – dann liefert die Funktion 3 als Ergebnis.

Oftmals ist es nicht erforderlich, in einem Freispeicher Werte von beliebigem Typ abzulegen. Dann kann man die benötigten Typen auch in einem Summentyp zusammenfassen und den Freispeicher als normale Liste modellieren. Um Raum für entsprechende Erweiterungen zu lassen, verzichten wir in den nachfolgend vorgestellten Calculi auf die Berücksichtigung des Freispeichers. Auf imperative Sprachmerkmale kommen wir erst später, im zweiten Teil dieser Arbeit zurück, wenn wir die Einbettung einer funktionalen Sprache in C++ diskutieren.

2.8 Weiterführende Literatur

Einführende Darstellungen zu den Grundlagen der universellen Algebra und der Ordnungstheorie findet man in nahezu allen Büchern, die sich mit der Semantik und den Typsystemen von Programmiersprachen beschäftigen. Exemplarisch sei auf die Werke von Pierce [142], Winskel [179], sowie dem eher mathematisch-motivierten Werk von Ihringer [77] verwiesen.

Termersetzungssysteme, die wir hier nur am Rande behandelt haben, werden von Baader und Nipkow in [11] ausführlich vorgestellt.

Einen umfangreichen Einstieg in die Theorie und Praxis der partiellen Auswertung findet sich in [88]. Einen schnellen Überblick bieten die Artikel von Consel und Danvy [36] und Jones [85, 87].

Die Bücher von Winskel [179] und Nielson [124] gelten als Standardwerke zur Semantik von Programmiersprachen. Typsysteme werden hier jedoch nur am Rande behandelt. Hier empfiehlt sich das Werk von Pierce, der in [142] einen guten Überblick über gängige Typsysteme liefert. Bruce diskutiert in [21] diverse Modellsprachen mit Fokus auf Objektorientierung.

Barendregts Abhandlung über die verschiedenen λ-Calculi gilt als eines der Standardwerke auf diesem Gebiet. Spezielle Darstellungen zu System **F** findet man bei Reynolds [146], Girard [55] und in [22]. Vertiefende Informationen zu System $\mathbf{F}_\omega$ bieten u.A. [55] und [142].

Obwohl wir im praktischen Teil davon Gebrauch machen werden, haben wir auf die Darstellung von objektorientierten Erweiterungen des λ-Kalküls verzichtet. Zur Diskussion der für uns primären Anwendung der strukturellen Typanalyse sind diese auch nicht zwingend erforderlich. Der interessierte Leser sei für einen Einstieg auf die Arbeiten von Bruce [20, 21], Bruce, Schuett und van Gent [23], sowie das Buch von Cardelli und Abadi [5] verwiesen. Cardellis und Mitchells „Operations on Record" [29] ist ein guter Ausgangspunkt, um sich mit der semantischen Modellierung von Objekten und Klassen zu beschäftigen.

Für die in den Beispielen verwendeten funktionalen Programmiersprachen Haskell und ML gibt es zahlreiche ein- und weiterführende Literatur. Einen kurzen Überblick über Haskell vermittelt der Artikel von Hudak [74]. Aufführlichere Darstellungen findet man z.B. bei Thompson [165] und natürlich im Haskell Standard [136]. ML wird von Paulson in [132] anhand von vielen praktischen Beispielen erklärt.

Kapitel 3

Strukturelle Typanalyse: System $\mathbf{F}^{\mathbf{SA}}_{\omega,1}$

Beim Vergleich von rekursiven Datentypen und Konstruktortypen (siehe Seite 50) haben wir gesehen, daß man zur sinnvollen Arbeit mit Konstruktortypen ein Äquivalent zur *case*-Analyse braucht.

In diesem Kapitel wollen wir eine entsprechende Erweiterung von System $\mathbf{F}_\omega$ vorstellen. Den Kern bilden zwei neue Sprachkonstrukte: `Typecase` und `typecase`. `Typecase` ermöglicht es, Typen durch strukturelle Induktion über den Namen von monomorphen Typen zu definieren, während `typecase` dasselbe für Terme möglich macht. Da beide Konstrukte auf der Analyse der Struktur von Typnamen aufbauen, spricht man auch von **struktureller Typanalyse**. Einen entsprechend erweiterten λ-Kalkül nennen wir System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ (SA steht für strukturelle Analyse).

Die Idee zu `Typecase` und `typecase` geht zurück auf Harper und Morrisett, die diese erstmals im Rahmen des λ_i^{ML}-Kalküls formalisiert haben (siehe [118, 67]). Unsere Darstellung lehnt sich zwar an diese Arbeiten an, wir werden jedoch ein paar Erweiterungen und Verallgemeinerungen vornehmen, um für den Vergleich mit der C++ Template-Metaprogrammierung besser gerüstet zu sein.

Aufgrund der starken Anlehnung an λ_i^{ML} verzichten wir auf die sonst übliche Untersuchung der theoretischen Merkmale von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$. Die von uns vorgenommenen Erweiterungen nehmen keinen Einfluß auf die zentralen Eigenschaften von λ_i^{ML}, so daß wir die Entscheidbarkeit der Typisierung und die Typerhaltung der Reduktion gewissermaßen von λ_i^{ML} „erben".

Nachdem wir `Typecase` und `typecase` anhand von Beispielen eingeführt haben, werden wir System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ formal beschreiben. Wir werden dann einen partiellen Auswerter für System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ vorstellen, der den Grundstein zu einer Übersetzung durch Typelimination bildet. Anschließend stellen wir Überladung und *pattern matching* als zwei für uns zentrale Anwendungsbeispiele für die strukturelle Typanalyse vor.

Am Ende des Kapitels geben wir einen kurzen Überblick über aktuelle Arbeiten und weiterführende Literatur zum Thema strukturelle Typanalyse.

3.1 System $\mathbf{F}^{\mathbf{SA}}_{\omega,1}$

Zentraler Bestandteil der strukturellen Typanalyse (auch **intensionelle Typanalyse** genannt) sind die Konstrukte `Typecase` und `typecase`.

Mit Hilfe von `Typecase` kann man Typen durch strukturelle Induktion über den Aufbau eines Typnamens definieren. Konzeptionell kann man sich `Typecase` als Typfunktion höherer Ordnung vorstellen, die für jeden n-stelligen Typkonstruktor eine n-stellige Typfunktion als Argument übernimmt.

In einer Haskell-ähnlichen Notation könnte `Typecase` etwa so implementiert sein:

$$
\begin{array}{l}
\texttt{Typecase}\ F_{\texttt{int}}\ F_{\texttt{bool}}\ F_{\rightarrow}\ F_C \cdots\ \alpha = \\
\quad \texttt{case}\ \alpha\ \texttt{of} \\
\quad\quad \begin{array}{lll}
\texttt{int} & -> & F_{\texttt{int}} \\
\texttt{bool} & -> & F_{\texttt{bool}} \\
\rightarrow (T_1, T_2) & -> & F_{\rightarrow}[T_1][T_1'][T_2][T_2'] \\
 & & \quad \texttt{where}\ T_1' = \texttt{Typecase}\ F_{\texttt{int}}\ F_{\texttt{bool}}\ F_{\rightarrow}\ F_C \cdots\ T_1 \\
 & & \quad\quad\quad T_2' = \texttt{Typecase}\ F_{\texttt{int}}\ F_{\texttt{bool}}\ F_{\rightarrow}\ F_C \cdots\ T_2 \\
\ldots & & \ldots
\end{array}
\end{array}
$$

Operationell traversiert `Typecase` den zu einem Typnamen gehörenden abstrakten Syntaxbaum (strukturelle Induktion) und manipuliert dessen Blätter (nullstellige Typkonstruktoren) und Knoten (mehrstellige Typkonstruktoren) entsprechend der zum Konstruktor gehörigen Funktion (die `Typecase` als Argument erhält).

An den Blättern wird einfach die zugehörige Funktion (bzw. Konstante) als Ergebnis zurückgegeben. Das Vorgehen an den Knoten ist ein wenig komplexer, um sowohl die *bottom-up*, als auch die *top-down* Berechnung von Typen zu erlauben.

Trifft `Typecase` z.B. auf einen Typ, der mit dem binären Konstruktor $\rightarrow$ gebildet wurde, so wird `Typecase` zunächst rekursiv auf die beiden Konstruktorargumente T_1 und T_2 angewendet (*bottom-up* Berechnung). Anschließend werden die dabei entstehenden Typen T_1' und T_2' zusammen mit den Originaltypen T_1 und T_2 an die Funktion $F_{\rightarrow}$ weitergereicht, wodurch sich die Möglichkeit der *top-down*-Berechnung ergibt.

Da das Ergebnis einer `Typecase`-Berechnung ein Typ ist, kann man `Typecase`-Terme selbst als Typen interpretieren. Die eigentliche Berechnung verläuft im Typsystem, wenn es um die Berechnung der Äquivalenz zweier Typen geht. Das Rekursionsmuster von `Typecase` ist integraler Bestandteil der Sprache System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ und kann vom Programmierer nicht manipuliert werden. Daraus erwachsen Vorteile bezüglich der Eigenschaften dieses Systems. Zum Beispiel kann garantiert werden, daß eine `Typecase`-Berechnung terminiert.

Im Vergleich zu λ_i^{ML} müssen `Typecase`-Terme in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ nicht für alle Konstruktoren einen Behandlungsfall aufweisen. So kann man zum Beispiel eine Funktion definieren, die nur mit bestimmten Konstruktortypen verträglich ist. Seien z.B. `C` und `D` zweistellige Konstruktoren, `N` ein nullstelliger Konstruktor und folgende Funktion gegeben:

$$
\begin{array}{l}
\texttt{zap} := \Lambda\delta :: K.\texttt{Typecase}\ \delta\ \texttt{of} \\
\quad\quad \begin{array}{ll}
\texttt{N}: & \texttt{int} \\
\texttt{C}: & \Lambda\alpha :: *.\Lambda\alpha' :: K. \\
 & \Lambda\beta :: *.\Lambda\beta' :: K. \\
 & \texttt{D}(\alpha', \beta')
\end{array}
\end{array}
$$

Die Funktion `zap` ersetzt im Typ δ alle Vorkommen von `N` durch `int` und alle Vorkommen von `C` durch `D`.

Offenbar ist `zap` nur auf Typen anwendbar, die ausschließlich aus den Konstruktoren `N` und `C` aufgebaut sind; für alle anderen Typen fehlt ein geeigneter Behandlungsfall.

Da die Zahl an Konstruktoren sehr hoch sein kann, bietet es sich an, einen `default`-*case* zu erlauben, der immer dann greift, wenn die anderen Muster nicht passen.

Wird dieser `default`-Fall gewählt, ist der Aufbau des zu untersuchenden Typs natürlich nicht bekannt und demzufolge kann die Traversierung des Baumes auch nicht fortgesetzt werden. Um dennoch Manipulationen zu ermöglichen, setzen wir voraus, daß im `default`-Fall eine Typfunktion mit Kind $* \Rightarrow K$ zur Verfügung gestellt wird und übergeben dieser im ersten Argument den Typnamen, zu dem kein anderer Behandlungsfall gefunden werden konnte.

Das Schlüsselwort `default` selbst fassen wir konzeptionell als nullstelligen Konstruktor auf; allerdings schließen wir (genau wie bei `void`) durch das Typsystem aus, daß Werte dieses Typs gebildet werden können.

Die strukturelle Dualität von Konstruktortypen und -werten legt nahe, eine Funktion `typecase` zu implementieren, mit der man Funktionen in Abhängigkeit von der Typstruktur definieren kann. Die Idee hinter `typecase` motivieren wir ebenfalls an einer Funktion, die wir in einer Haskell-ähnlichen Notation angeben:

$$
\begin{array}{l}
\texttt{typecase}\ f_{\texttt{int}}\ f_{\texttt{bool}}\ f_{\rightarrow}\ f_C \cdots\ \alpha = \\
\quad\ \texttt{case}\ \alpha\ \texttt{of} \\
\quad\quad \begin{array}{lcl}
\texttt{int} & -> & f_{\texttt{int}} \\
\texttt{bool} & -> & f_{\texttt{bool}} \\
\rightarrow (T_1, T_2) & -> & f_{\rightarrow}[T_1]\ f_1\ [T_2]\ f_2 \\
 & & \quad \texttt{where}\ f_1 = \texttt{typecase}\ f_{\texttt{int}}\ f_{\texttt{bool}}\ f_{\rightarrow}\ f_C \cdots\ T_1 \\
 & & \qquad\quad f_2 = \texttt{typecase}\ f_{\texttt{int}}\ f_{\texttt{bool}}\ f_{\rightarrow}\ f_C \cdots\ T_2 \\
\ldots & & \ldots
\end{array}
\end{array}
$$

Strukturell sind sich `Typecase` und `typecase` sehr ähnlich. Der wesentliche Unterschied liegt im berechneten Objekt und im Typ der Funktionen, die als Argumente übernommen werden.

Trifft `typecase` auf ein Blatt, so wird die zum Konstruktor gehörige Funktion als Ergebnis zurückgeliefert. Beim Besuch eines Knotens werden durch rekursiven Aufruf von `typecase` zunächst Funktionen zur Manipulation der Konstruktorkomponenten generiert. Funktionen zur Manipulation von mehrstelligen Konstruktoren wird im Wechsel der Typ der nicht-manipulierten Komponente, sowie eine Funktion zur Manipulation der Komponenten übergeben.

Betrachten wir ein Anwendungsbeispiel. Ein `typecase`-Term zur Berechnung der Schachtelungstiefe eines Typausdrucks, der aus dem einstelligen Konstruktor `S` und dem nullstelligen Konstruktor `N` konstruiert wurde, könnten wir folgendermaßen realisieren:

$$
\begin{array}{l}
\Lambda\delta :: *.\texttt{typecase}\ \delta\ \ \langle \Lambda\gamma.\texttt{int} \rangle\ \texttt{of} \\
\quad \begin{array}{ll}
\texttt{N}: & 0 \\
\texttt{S}: & \Lambda\alpha :: *.\lambda v : \texttt{int} \\
 & \quad 1 + v
\end{array}
\end{array}
$$

Syntaktisch ist `typecase` aufgebaut wie sein Typäquivalent. Hinzugekommen ist eine Berechnungsvorschrift für den Ergebnistyp, den wir in spitzen Klammern hinter den Typparameter schreiben. Auf die Bedeutung dieses Ausdrucks gehen wir später ein.

Der Typ `N` hat die Schachtelungstiefe Null, während die Schachtelungstiefe eines Wertes vom Typ $\mathtt{S}(T)$ um eins größer ist als die Schachtelungstiefe des Typs T. Der Ausführungsmechanismus zu `typecase` sorgt dafür, daß eben diese Schachtelungstiefe im Wert v zur Verfügung steht.

Betrachten wir ein weiteres Beispiel, um uns die Bedeutung der Berechnungsvorschrift für den Ergebnistyp klar zu machen. Sei wieder `C` ein zweistelliger und `N` ein nullstelliger Konstruktor. Wie wir bereits in Abschnitt 2.7.2 gezeigt haben, kann man mit `C` und `N` Polytyplisten konstruieren. So beschreibt der Typ

$$\mathtt{C}(\mathtt{int}, \mathtt{C}(\mathtt{bool}, \mathtt{N}))$$

Listen aus einem `int`- und einem `bool`-Wert.

Angenommen, wir wollen in einer solchen Liste alle `int`-Werte durch boolesche Werte ersetzten und zwar so, daß wir `false` eintragen, wo vorher ein Wert größer Null war, und `true`, wo ein Wert kleiner oder gleich Null steht.

Diese Operation ändert sowohl den Typ, als auch den Wert. Wir starten mit der Berechnung des geänderten Typs. Dazu können wir folgenden `Typecase`-Term bereitstellen:

$$\begin{array}{lll}
\mathtt{MkBool} := \Lambda\gamma :: *.\mathtt{Typecase}\ \gamma\ \mathtt{of} & & \\
\qquad\qquad\qquad

`MkBool`$[\alpha]$ und `MkBool`$[\beta]$. Der Ergebnistyp der Funktion T_R ergibt sich durch Anwendung von `MkBool`$[$`C`$(\alpha, \beta)]$. Das heißt, wir können den Typ, den die Funktion eines bestimmten Behandlungsfalles haben muß, allein aus der Funktion zur Berechnung des Ergebnistyps und der Stelligkeit des Konstruktors ermitteln.

Wenn wir Konstruktoren selbst als Typfunktionen interpretieren, können wir leicht einen Operator definieren, der uns ausgehend von einem Konstruktorsymbol einen geeigneten Typ konstruiert; der `default`-Konstruktor erfährt eine Sonderbehandlung:

$$\begin{aligned}
&\langle F\rangle(T :: *) = F[T]\\
&\langle F\rangle(\texttt{default} :: *) = \forall\alpha.F[\alpha]\\
&\langle F\rangle(T :: K_1 \Rightarrow K_2) = \forall\alpha :: K_1.\langle F\rangle(\alpha :: K_1) \to \langle F\rangle(T(\alpha) :: K_2)
\end{aligned}$$

Zur Typüberprüfung von `typecase` können wir den erwarteten Typ mit diesem Operator für jeden Zweig berechnen, um dann zu untersuchen, ob er äquivalent zum Typ des zugehörigen Terms ist.

Betrachten wir abschließend eine Funktion zur Berechnung der Länge einer Polytypliste:

$$\begin{aligned}
\texttt{dynLength} := \lambda\delta :: *.\texttt{typecase}\ \delta\ \ &\langle\Lambda\gamma :: *.\gamma \to \texttt{int}\rangle\ \texttt{of}\\
\texttt{default}:\ \ &\Lambda\alpha :: *.\lambda x : \alpha.0\\
\texttt{N}:\ \ &\lambda x : N.0\\
\texttt{C}:\ \ &\Lambda\alpha :: *.f_0 : \alpha \to \texttt{int}.\\
&\Lambda\beta :: *.f_1 : \beta \to \texttt{int}.\\
&\lambda l : \texttt{C}(\alpha, \beta).1 + f_1(l.1)
\end{aligned}$$

Wird `dynLength` auf einen Listentyp angewendet, so wird durch den `typecase` Mechanismus eine Funktion generiert, die die Länge eines passenden Listenwertes berechnet. Die eigentliche Berechnung der Listenlänge findet zur Laufzeit statt, wenn die so generierte Funktion auf ein konkretes Argument angewendet wird.

Da die Länge einer Liste allein von ihrer Struktur bestimmt wird und wir diese aus dem Typnamen ablesen können, kann `dynLength` optimiert werden zu:

$$\begin{aligned}
\texttt{statLength} := \lambda\delta :: *.\texttt{typecase}\ \delta\ \ &\langle\Lambda\gamma :: *.\texttt{int}\rangle\ \texttt{of}\\
\texttt{default}:\ \ &0\\
\texttt{N}:\ \ &0\\
\texttt{C}:\ \ &\Lambda\alpha :: *.v_0 : \texttt{int}.\\
&\Lambda\beta :: *.v_1 : \texttt{int}.\\
&1 + v_1
\end{aligned}$$

Die Anwendung von `statLength` auf einen Listentyp führt zu einem System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Ausdruck, der nur Konstanten erhält. Dieser könnte von einem partiellen Auswerter reduziert werden, so daß mit `statLength` die Länge einer Liste bereits zur Übersetzungszeit zur Verfügung steht.

Anmerkung 3.1. Die `default`-Fälle sind jeweils erforderlich, da `Typecase` und `typecase` strikt realisiert sind – die Berechnung wird grundsätzlich in jedem Zweig eines Konstruktortyps durchgeführt. ◊

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

$M ::=$		**Monomorphe Typen**
	$\cdots$	
	$\texttt{Typecase}\ M\ \texttt{of}\ R_1 \cdots R_n$	Typecase-Typ
$R ::=$		**Type-Typecase**
	$\texttt{C} : T$	Konstruktor-Fall
	$\texttt{default} : T$	Default-Fall
$R_t ::=$		**Value-Typecase**
	$\texttt{C} : t$	Konstruktor-Fall
	$\texttt{default} : t$	Default-Fall
$t ::=$		**Terme**
	$\cdots$	
	$\texttt{typecase}\ t\ \langle \Lambda\alpha.T \rangle\ \texttt{of}\ Rt_1 \cdots Rt_n$	Typecase-Term

Abbildung 3.1: Syntaktische Erweiterungen von System $\mathbf{F}_\omega$ zur Unterstützung von struktureller Typanalyse. Zur Erinnerung: Die Grundtypen `int` und `bool` und der Typkonstruktor $\rightarrow$ sind Bestandteil der Signatur $\Sigma_{\mathcal{C}}$.

3.1.1 Syntax

Abbildung 3.1 zeigt die Syntax von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ als Erweiterung zu System $\mathbf{F}_\omega$.

Eine Sequenz von n **Type-Typecases** schreiben wir auch als $(\texttt{C}_i : T_i)^{i\in\{1,\ldots,n\}}$ und analog eine Sequenz von n **Value-Typecases** als $(\texttt{C}_i : t_i)^{i\in\{1,\ldots,n\}}$.

3.1.2 Kinding und Typäquivalenz

Zur Realisierung von Operator $\langle\cdot\rangle$ müssen wir Konstruktoren als Funktionen über Typen interpretieren. Für diese gibt es jedoch keine Funktionsvorschrift – die vollständige Applikation eines Konstruktors zieht keine Berechnung nach sich, sondern stellt den zu berechnenden Wert selbst dar.

$\boxed{\Gamma \vdash T :: K}$ System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

$$\frac{\sigma_{\mathcal{C}}(\texttt{C}) = 1}{\Gamma \vdash \texttt{C} :: * \Rightarrow *}\ \textbf{(K-Cons-0)} \qquad \frac{\sigma_{\mathcal{C}}(\texttt{C}) = n}{\Gamma \vdash \texttt{C} :: (\Rightarrow)^n *}\ \textbf{(K-Cons-n)}$$

$$\frac{\Gamma \vdash \texttt{C} :: (\Rightarrow)^n * \quad \Gamma \vdash T :: *}{\Gamma \vdash \texttt{C}(T) :: (\Rightarrow)^{n-1} *}\ \textbf{(K-Cons-App)}$$

$$\frac{\begin{array}{c}\Gamma \vdash M :: * \quad \forall i : \Gamma \vdash_R M_i : T_i :: K \\ \exists^1 \texttt{C}_i \in \{\texttt{C}_1, \cdots, \texttt{C}_n\} : \Gamma \vdash M \simeq \texttt{C}_i \ \vee\ \texttt{default} \in \{\texttt{C}_1, \cdots, \texttt{C}_n\}\end{array}}{\Gamma \vdash \texttt{Typecase}\ M\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} :: K}\ \textbf{(K-Typecase)}$$

Abbildung 3.2: Kinding in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

Die Schreibweise $\texttt{C}(T_1, \cdots, T_n)$ verstehen wir von nun als syntaktischen Zucker für $\texttt{C}(T_1) \cdots (T_n)$[1]

[1]Wir verzichten darauf, die abstrakte Syntax so anzupassen, daß sie auch partiell applizierte Konstruktoren

und schreiben ferner $(\Rightarrow)^n$, um die n-fache Anwendung des Kind-Konstruktors $\Rightarrow$ zu beschreiben. Zum Beispiel entspricht $(\Rightarrow)^3 *$ dem Kind $* \Rightarrow * \Rightarrow * \Rightarrow *$. Wir setzen $(\Rightarrow)^0 K = K$.

Abbildung 3.2 zeigt die Kinding-Regeln für System $\mathbf{F}^{SA}_{\omega,1}$.

Bei der Zuweisung einer Gattung zu einem `Typecase`-Term setzen wir zwei Hilfsrelationen ein, die uns eine kompaktere Schreibweise erlauben (siehe Abbildung 3.3).

System $\mathbf{F}^{SA}_{\omega,1}$

$\boxed{\Gamma \vdash_R \mathtt{C} : T :: K}$

$$\frac{\Gamma \vdash \mathtt{C} :: * \quad \Gamma \vdash T :: K}{\Gamma \vdash_R \mathtt{C} : T :: K} \textbf{ (R-Cons-0)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{C} :: (\Rightarrow)^n * \\ \Gamma \vdash T :: (\Rightarrow)^n (* \Rightarrow K) \Rightarrow K\end{array}}{\Gamma \vdash_R \mathtt{C} : T :: K} \textbf{ (R-Cons-n)}$$

$$\frac{\Gamma \vdash T :: * \Rightarrow K}{\Gamma \vdash_R \mathtt{default} : T :: K} \textbf{ (R-Default)}$$

$\boxed{\Gamma \vdash \mathtt{C} \simeq T_2}$

$$\frac{\Gamma \vdash T \equiv T'}{\Gamma \vdash T \simeq T'} \textbf{ (M-Equiv)}$$

$$\frac{\Gamma \vdash \mathtt{C}'(T) :: * \quad \Gamma \vdash \mathtt{C}(T) \equiv \mathtt{C}'(T)}{\Gamma \vdash \mathtt{C}(T) \simeq \mathtt{C}'} \textbf{ (M-Cons-1)}$$

$$\frac{\Gamma \vdash \mathtt{C}'(T_1, T_2) :: * \quad \Gamma \vdash \mathtt{C}(T_1, T_2) \equiv \mathtt{C}'(T_1, T_2)}{\Gamma \vdash \mathtt{C}(T_1, T_2) \simeq \mathtt{C}'} \textbf{ (M-Cons-2)}$$

$\cdots$ usw. bis **M-Cons-n**:

$$\frac{\Gamma \vdash \mathtt{C}'(T_1, \cdots, T_n) :: * \quad \Gamma \vdash \mathtt{C}(T_1, \cdots, T_n) \equiv \mathtt{C}'(T_1, \cdots, T_n)}{\Gamma \vdash \mathtt{C}(T_1, \cdots, T_n) \simeq \mathtt{C}'} \textbf{ (M-Cons-n)}$$

Abbildung 3.3: Relationen $\vdash_R$, zur Überprüfung der Typisierbarkeit von Behandlungsfällen in `Typecase`-Termen, und $\simeq$ zum Matching von Konstruktoren und Typtermen.

Mit der Relation $\vdash_R$ prüfen wir die Korrektheit und die Gattung, zu der der Ergebnistyp eines Musters gehört. Wir schreiben $\Gamma \vdash_R \mathtt{C} : T :: K$, wenn der Type-Typecase $\mathtt{C} : T$ unter den Voraussetzungen der Typisierungsumgebung Γ korrekt ist und zu einem Typ der Gattung K führt.

Für nullstellige Konstruktoren ist dies der Fall, wenn sie korrekte Monotypen sind, und der zugehörigen Typfunktion die Gattung K zugewiesen werden kann (sie also eine korrekte Typfunktion ist). Von Behandlungsfällen für n-stellige Konstruktoren wird erwartet, daß die Typfunktion für jedes Konstruktorargument im Wechsel einen Monotyp (Gattung $*$) und einen Typ der Gattung K erwartet und im Ergebnis zu einem Typ der Gattung K führt (es sind also auch Polytypen als Ergebnis erlaubt).

Die Relation $\simeq$ (siehe Abbildung 3.3) beschreibt die Zuordnung eines Typs zu einem Konstruktor und damit zu einem bestimmten Behandlungsfall, wobei natürlich die Äquivalenz von Typen berücksichtigt werden muß. $\Gamma \vdash T \simeq \mathtt{C}$ bedeutet, daß unter den Voraussetzungen

erlaubt. Die Vorstellung, daß ein Konstruktor eine Funktion ist, wird nur intern, auf der Ebene von Kinding, Typsystem und Typäquivalenz verfolgt.

in Γ gilt, daß der Typ T zum Konstruktor `C` paßt. Beachte: Der Konstruktor `default` bleibt unberücksichtigt, so daß von $\simeq$ nur reguläre Konstruktoren gematcht werden.

`Typecase`-Terme werden als gültige Typen erkannt, wenn der zu analysierende Typ ein gültiger Monotyp ist, es genau einen Behandlungsfall gibt, der zu diesem Typ paßt, oder ein `default`-case vereinbart wurde, und alle Behandlungsfälle zu einem Typ derselben Gattung K führen (siehe Regel **K-Typecase** in Abbildung 3.2).

Als Typ ist `Typecase` äquivalent zu anderen Typen, wobei sich äquivalente Typen aus der „Berechnung" von `Typecase` ergeben.

$\boxed{\Gamma \vdash T_1 \equiv T_2}$ System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

$$\frac{\Gamma \vdash T \equiv T'}{\Gamma \vdash \texttt{Typecase}\ T\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} \equiv \texttt{Typecase}\ T'\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}}} \quad \textbf{(Q-TC-Cong)}$$

$$\frac{\begin{array}{c}\exists i \in \{1,\ldots,m\} : \Gamma \vdash \texttt{C} \simeq \texttt{C}_i \\ \Gamma \vdash \texttt{Typecase C of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} :: *\end{array}}{\Gamma \vdash \texttt{Typecase C of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} \equiv T_i} \quad \textbf{(Q-TC-Cons-0)}$$

$$\frac{\begin{array}{c}\exists i \in \{1,\ldots,m\} : \Gamma \vdash \texttt{C}(M) \simeq \texttt{C}_i \\ \Gamma \vdash \texttt{Typecase C}(M)\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} :: *\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{Typecase C}(M)\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} \equiv \\ \qquad T_i[M][\texttt{Typecase}\ M\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}}]\end{array}} \quad \textbf{(Q-TC-Cons-1)}$$

$\cdots$ usw. bis **E-TC-Cons-n**:

$$\frac{\begin{array}{c}\exists i \in \{1,\ldots,m\} : \Gamma \vdash \texttt{C}(M_1,\cdots,M_n) \simeq \texttt{C}_i \\ \Gamma \vdash \texttt{Typecase C}(M_1,\cdots,M_n)\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} :: *\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{Typecase C}(M_1,\cdots,M_n)\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} \equiv \\ \qquad T_i[M_1][\texttt{Typecase}\ M_1\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}}] \\ \qquad\quad \vdots \\ \qquad [M_n][\texttt{Typecase}\ M_n\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}}]\end{array}} \quad \textbf{(Q-TC-Cons-n)}$$

$$\frac{\begin{array}{c}\nexists i \in \{1,\ldots,m\} : \Gamma \vdash T \simeq \texttt{C}_i \\ \texttt{default} \in \{\texttt{C}_1,\ldots,\texttt{C}_m\} \\ \Gamma \vdash \texttt{Typecase}\ T\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} :: *\end{array}}{\Gamma \vdash \texttt{Typecase}\ T\ \texttt{of}\ (\texttt{C}_i : T_i)^{i\in\{1,\ldots,m\}} \equiv T_i[T]} \quad \textbf{(Q-TC-Default)}$$

Abbildung 3.4: Typäquivalenz in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

Die Regeln **Q-TC-Cons-0** bis **Q-TC-Cons-N** können nur dann Anwendung finden, wenn es genau einen passenden Behandlungsfall für den zu untersuchenden Monotyp gibt (siehe **K-Typecase**: $\simeq$ läßt `default`-Behandlungsfälle außer acht). Der `default`-Fall greift nur dann, wenn es keinen passenden Behandlungsfall gibt, dafür aber ein `default`-Type-Typecase vereinbart wurde.

Beachte: konnte ein passender Behandlungsfall gefunden werden, so bekommt die zugehörige Typfunktion T_i für jede Konstruktorkomponente sowohl den Originaltyp M_i, als auch das Ergebnis der Anwendung von `Typecase` auf M_i als Argument übergeben. `Typecase` ist damit

inhärent rekursiv.

3.1.3 Typsystem

Wie bereits erwähnt, prüfen wir `typecase`-Anweisungen, indem wir mit Hilfe des Operators $\langle F \rangle$ zu jedem Konstruktor einen erwarteten Ergebnistyp konstruieren, den zugehörigen Typ des Rückgabewertes berechnen und abschließend beide Typen auf Typäquivalenz hin untersuchen.

$\boxed{\Gamma \vdash t : T}$ System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

$$\frac{\begin{array}{c}\Gamma \vdash T :: *\\ \Gamma \vdash F :: * \Rightarrow *\\ \exists^1 i \in \{1,\cdots,m\} : \Gamma \vdash T \simeq \mathtt{C}_i \vee \mathtt{default} \in \{\mathtt{C}_1,\cdots,\mathtt{C}_m\}\\ \forall i : \Gamma \vdash t_i : \langle F \rangle(\mathtt{C}_i)\end{array}}{\Gamma \vdash \mathtt{typecase}\ T\ \ \langle F \rangle\ \mathtt{of}\ (\mathtt{C}_i : t_i)^{i \in \{1,\ldots,m\}} : F[T]}\ \textbf{(T-typecase)}$$

Abbildung 3.5: Typsystem zu System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

Die Sonderbehandlung des `default`-Falls ist hier nicht erforderlich. Da wir `default` als nullstelligen Konstruktor ansehen, ist $\langle F \rangle(\mathtt{default})$ definiert und mit **K-Typecase** und **Q-TC-Default** haben wir sichergestellt, daß es sich bei dem Ergebnistyp tatsächlich um eine Typfunktion handelt.

3.1.4 Operationelle Semantik

Die in den einzelnen Behandlungsfällen definierten Funktionen bekommen für jedes Konstruktorargument sowohl den Argumenttyp, als auch das Ergebnis der rekursiven Anwendung von `typecase` auf diese Komponente, als Argument übergeben.

Paßt keiner der Behandlungsfälle und wurde ein `default`-Fall vereinbart, so wird nur der zu untersuchende Typ an die Behandlungsfunktion weitergeleitet.

Die Umsetzung von `Typecase` und `typecase` ließe sich noch optimieren. In der hier vorgestellten Version werden *bottom-up* Werte grundsätzlich berechnet – auch dann, wenn sie im zugehörigen Term nicht gebraucht werden. Ein entsprechender Test auf Verwendung einer Variablen läßt sich sehr leicht in die Semantikregeln integrieren, allerdings stellt sich dann die Frage, welchen Wert man als Ersatz an die jeweilige Funktion übergibt.

Als konsistente Lösung bietet sich eine Erweiterung von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ um Subtyping an, wobei man einen kleinsten Typ $\bot$ ergänzt. Dann kann man das nicht benötigte Argument einfach durch $\bot$ bzw. eine Instanz von $\bot$ ersetzen.

Der Vorteil dieser Erweiterung läge darin, daß man in vielen Fällen auf den `default`-Fall verzichten kann (zum Beispiel bei der Längenberechnung für Listen auf Seite 61).

3.2 Eigenschaften von System $\mathbf{F}^{\mathbf{SA}}_{\omega,1}$

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ist eine Erweiterung von λ_i^{ML}. Im Vergleich zur Originalarbeit von Morrisett und Harper ist System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ nicht auf drei Konstruktortypen beschränkt. Darüber hinaus müssen

$t_1 \to t_2$

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

$$\frac{\exists i \in \{1,\ldots,m\} : \vdash \mathtt{C}_i \simeq \mathtt{C}}{\mathtt{typecase\ C}\ \langle F\rangle\ \mathtt{of}\ (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}} \to t_i} \quad \textbf{(E-TCase-0)}$$

$$\frac{\exists i \in \{1,\ldots,m\} : \vdash \mathtt{C}(T) \simeq \mathtt{C}}{\begin{array}{l}\mathtt{typecase\ C}(T)\ \langle F\rangle\ \mathtt{of}\ (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}} \to \\ \quad t_i[T](\mathtt{typecase}\ T\langle F\rangle\ \mathtt{of}\ (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}})\end{array}} \quad \textbf{(E-TCase-1)}$$

$\cdots$ usw. bis **E-T-Case-N**:

$$\frac{\exists i \in \{1,\ldots,m\} : \vdash \mathtt{C}(T_1,\cdots,T_n) \simeq C_i}{\begin{array}{l}\mathtt{typecase\ C}(T_1,\cdots,T_n)\ \langle F\rangle\ \mathtt{of}\ (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}} \to \\ \quad t_i[T_1](\mathtt{typecase}\ T_1\langle F\rangle\ \mathtt{of}\ (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}}) \\ \quad \vdots \\ \quad [T_n](\mathtt{typecase}\ T_n\langle F\rangle\ \mathtt{of}\ (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}})\end{array}} \quad \textbf{(E-TCase-n)}$$

$$\frac{\begin{array}{c}\not\exists i \in \{1,\ldots,m\} : \vdash T \simeq C_i \\ \mathtt{default} \in \{\mathtt{C}_1,\ldots,\mathtt{C}_m\}\end{array}}{\begin{array}{l}\mathtt{typecase\ C}(T_1,\cdots,T_n)\langle F\rangle\ \mathtt{of}\ (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}} \to \\ \quad t_i[\mathtt{C}_i(T_1,\cdots,T_n)]\end{array}} \quad \textbf{(E-TCase-Default)}$$

Abbildung 3.6: Operationelle Semantik von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

in einem `Typecase`- oder `typecase`-Term nicht alle Konstruktoren bei der Fallunterscheidung berücksichtigt werden. Neu ist auch die Einführung des `default`-Falls.

Wie man sich leicht überlegt, bleibt die Entscheidbarkeit des Typsystems in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ erhalten: `Typecase` und `typecase` sind zwar rekursiv definiert, allerdings ist in beiden Fällen garantiert, daß die Rekursion terminiert, da System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ keine unendlichen Typen unterstützt.

Ein expliziter Beweis zur Entscheidbarkeit des Typsystems von λ_i^{ML} findet sich bei Morrisett [118, Seite 46 ff.]. Ebenfalls bei Morrisett findet man Beweise zu Fortschritt und Typerhaltung der Reduktionsrelation.

Der Nachweis des entscheidbaren Typsystems verläuft ähnlich wie für System $\mathbf{F}_\omega$. Aus den Regeln zur Beschreibung von äquivalenten Typen konstruiert man eine Reduktionsrelation, deren reflexiver und transitiver Abschluß gerade der Typäquivalenzrelation entspricht. Für diese weist man nach, daß sie konfluent und terminierend ist, so daß man Typäquivalenz auf den Vergleich von Normalformen reduzieren kann. Eine entsprechend erweiterte Reduktionsrelation für Typen ($\rightsquigarrow$) findet sich in Abbildung 3.7.

Es sei vorweggenommen, daß System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ für unsere Zwecke nicht ausreicht. Die Entscheidbarkeit des Typsystems bedeutet, daß wir in Typausdrücken keinen Fixpunktkombinator verwenden können. Damit ist es nicht möglich, partielle oder rekursive Typfunktionen zu definieren – unsere Typ-Sprache ist noch nicht Turing-mächtig.

Im Vergleich zu System $\mathbf{F}$ und System $\mathbf{F}_\omega$ werden Laufzeitentscheidungen in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ durch `typecase` sehr wohl abhängig vom Typ. Lassen wir separate Übersetzung außer acht, können wir jedoch – wie wir gleich zeigen werden – einen partiellen Auswerter realisieren, der alle `typecase`-Anweisungen aus einem Programm eliminiert. Der Term, der bei der partiel-

len Auswertung entsteht, kann dann in einen äquivalenten Term des ungetypten λ-Kalküls übersetzt werden, so daß auch System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ durch Typelimination übersetzt werden kann.

$T_1 \rightsquigarrow T_2$ System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

$$\frac{T_1 \rightsquigarrow T_1' \quad \cdots T_n \rightsquigarrow T_n'}{C(T_1, \cdots, T_n) \rightsquigarrow C(T_1', \cdots, T_n')} \textbf{ (QR-Cons)}$$

$$\frac{\begin{array}{c} T \rightsquigarrow C_i(T_1', \cdots, T_n') \\ \texttt{Typecase}\ T_1'\ \texttt{of}\ (C_i : T_i)^{i\in\{1,\ldots,m\}} \rightsquigarrow T_{r1} \\ \vdots \\ \texttt{Typecase}\ T_n'\ \texttt{of}\ (C_i : T_i)^{i\in\{1,\ldots,m\}} \rightsquigarrow T_{rn} \\ T_i[T_1'][T_{r1}]\cdots[T_n'][T_{rm}] \rightsquigarrow T_r \end{array}}{\texttt{Typecase}\ T\ \texttt{of}\ (C_i : T_i)^{i\in\{1,\ldots,m\}} \rightsquigarrow T_r} \textbf{ (QR-Typecase)}$$

Abbildung 3.7: Erweiterung von $\rightsquigarrow$ zur Berücksichtigung von Konstruktortypen und `Typecase` (Auszug).

3.3 Partielle Auswertung von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

Den partiellen Auswerter zur Elimination von `typecase` und `Typecase` beschreiben wir als Funktion $\mathcal{T}$, die System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Terme auf System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Terme abbildet (siehe Abbildung 3.8).

Wir setzen voraus, daß Terme, die $\mathcal{T}$ als Argument erhält, den Typechecker für System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ passiert haben, abgeschlossen sind und der Typ des Terms ein Monotyp ist.

Terme, die ihren Ursprung in $\lambda^{\mathtt{fix}}$ haben, läßt $\mathcal{T}$ nahezu unangetastet. Lediglich die Typinformation des Arguments einer λ-Abstraktion wird ausgewertet; d.h. mit Hilfe von $\rightsquigarrow$ zu einer Normalform reduziert. Dieser Schritt ist notwendig, da an dieser Stelle nicht nur ein monomorpher Typ, sondern ebenso gut eine Typ-Typ-Applikation oder ein `Typecase`-Term stehen könnte.

In System $\mathbf{F}$ kamen typabhängige Terme hinzu. $\mathcal{T}$ vollzieht die Reduktion der Term-Typ-Applikation in Anlehnung an die Semantikregeln **E-TAppAbs** und **E-TApp** (siehe Abbildung 2.7 auf Seite 42). Typvariablen läßt der partielle Auswerter unangetastet. Da wir vorausgesetzt haben, daß der an $\mathcal{T}$ übergebene Term abgeschlossen und monomorph ist, werden Typvariablen spätestens bei der partiellen Auswertung von Term-Typ-Applikationen eliminiert.

Mit System $\mathbf{F}_\omega$ wurde das Konzept der Typäquivalenz eingeführt. Typen werden von $\mathcal{T}$ durch ihre Normalform bezüglich der Reduktionsrelation $\rightsquigarrow$ ersetzt. Das hat natürlich zur Folge, daß `Typecase`-Terme und Typ-Typ-Applikationen „ausgerechnet" werden.

Bei der Behandlung von `typecase` werden nicht nur Typen reduziert. Betrachten wir die Arbeitsweise von $\mathcal{T}$ am Beispiel der Regel **E-TCase-1**:

$$\frac{\exists i \in \{1,\ldots,m\} : \vdash C(T) \simeq C}{\begin{array}{l}\texttt{typecase}\ \texttt{C}(T)\ \langle F\rangle\ \texttt{of}\ (\texttt{C}_i : t_i)^{i\in\{1,\ldots,m\}} \rightarrow \\ \quad t_i[T](\texttt{typecase}\ T\langle F\rangle\ \texttt{of}\ (\texttt{C}_i : t_i)^{i\in\{1,\ldots,m\}})\end{array}} \textbf{ (E-TCase-1)}$$

$\lambda^{\texttt{fix}}$ $\mathcal{T}$

$$
\begin{array}{rcl}
\mathcal{T}[\![C]\!] & = & C \quad \text{falls } \sigma_{\mathcal{C}} = 0 \\
\mathcal{T}[\![c]\!] & = & c \\
\mathcal{T}[\![x]\!] & = & x \\
\mathcal{T}[\![t_1\ t_2]\!] & = & \mathcal{T}[\![\ \mathcal{T}[\![t_1]\!]\mathcal{T}[\![t_2]\!]\ ]\!] \\
\mathcal{T}[\![(\lambda x : T.t)]\!] & = & \lambda x : \mathcal{T}[\![T]\!].\mathcal{T}[\![t]\!] \\
\mathcal{T}[\![(t_1 + t_2)]\!] & = & \mathcal{T}[\![t_1]\!] + \mathcal{T}[\![t_2]\!] \\
\mathcal{T}[\![\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3]\!] & = & \texttt{if } \mathcal{T}[\![t_1]\!] \texttt{ then } \mathcal{T}[\![t_2]\!] \texttt{ else } \mathcal{T}[\![t_3]\!] \\
\mathcal{T}[\![!t]\!] & = & !\mathcal{T}[\![t]\!] \\
\mathcal{T}[\![\texttt{fix } t]\!] & = & \texttt{fix } \mathcal{T}[\![t]\!]
\end{array}
$$

System $\mathbf{F}$

$$
\begin{array}{rcl}
\mathcal{T}[\![\alpha]\!] & = & \alpha \\
\mathcal{T}[\![t[T]]\!] & = & \mathcal{T}[\![\ \mathcal{T}[\![t]\!]\ [\mathcal{T}[\![T]\!]]\ ]\!] \\
\mathcal{T}[\![(\Lambda\alpha :: K.t)[T]]\!] & = & \mathcal{T}[\![t[T/\alpha]]\!] \\
\mathcal{T}[\![\Lambda\alpha :: K.t]\!] & = & \Lambda\alpha :: K.\mathcal{T}[\![t]\!]
\end{array}
$$

System $\mathbf{F}_\omega$

$$
\begin{array}{rcl}
\mathcal{T}[\![T]\!] & = & T \Downarrow_{\rightsquigarrow} \\
\mathcal{T}[\![\mathtt{C}(t_1, \cdots, t_n)]\!] & = & \mathtt{C}(\mathcal{T}[\![t_1]\!], \cdots, \mathcal{T}[\![t_n]\!]) \\
\mathcal{T}[\![t.i]\!] & = & \mathcal{T}[\![\ \mathcal{T}[\![t]\!].i\ ]\!] \\
\mathcal{T}[\![v.i]\!] & = & v.i \\
\mathcal{T}[\![\mathtt{C}(t_1, \cdots, t_n).i]\!] & = & \mathcal{T}[\![t_i]\!]
\end{array}
$$

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

$$\mathcal{L}[\![(\lambda x : T.t_1)(t_2)]\!] \quad = \quad \mathcal{T}[\![t_1[t_2/x]]\!]$$

$\mathcal{T}[\![\texttt{typecase C of } (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}}]\!] = \mathcal{T}[\![t_i]\!]$
falls $\exists \mathtt{C}_i :: * \in \{\mathtt{C}_1, \cdots, \mathtt{C}_m\} : \vdash \mathtt{C} \simeq \mathtt{C}_i$
$\mathcal{T}[\![\texttt{typecase } M \texttt{ of } (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}}]\!] = \mathcal{T}[\![t_i[M]]\!]$
falls $\nexists \mathtt{C}_i \in \{\mathtt{C}_1, \cdots, \mathtt{C}_m\} : \vdash M \simeq \mathtt{C}_i \ \wedge\ \exists i \in \{1, \cdots, n\} : \mathtt{C}_i = \texttt{default}$
$\mathcal{T}[\![\texttt{typecase C}(M_1, \cdots, M_n) \texttt{ of } (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}}]\!] =$
$\mathcal{L}[\![\mathcal{T}[\![\cdots \mathcal{L}[\![\mathcal{T}[\![\ \cdots\ \mathcal{L}[\![\mathcal{T}[\![t_i[M_1]]\!](\mathcal{T}[\![\texttt{typecase } M_1 \texttt{ of } (\mathtt{C}_i : t_i)^{i\in\{1,\ldots,m\}}]\!])]\!]$
$\vdots$
$[M_n]]\!](\mathcal{T}[\![\texttt{typecase } M_n \texttt{ of } (\mathtt{C}_i : T_i)^{i\in\{1,\ldots,m\}}]\!])]\!] \cdots]\!]$
$]\!]$
falls $\exists \mathtt{C}_i :: (\Rightarrow)^n * \in \{\mathtt{C}_1, \cdots, \mathtt{C}_m\} : \vdash \mathtt{C}(T_1, \cdots, T_n) \simeq \mathtt{C}_i$

Abbildung 3.8: Partieller Auswerter für System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

Die Typapplikation $t_i[T]$ wird durch Anwenden von $\mathcal{T}$ direkt reduziert – dabei entstehe eine Funktion t'_i, die das Ergebnis der rekursiven Anwendung von `typecase` auf T als Argument erwartet. Der partielle Auswerter fährt mit der Reduktion des `typecase`-Terms fort, um anschließend mit $\mathcal{L}$ das Ergebnis in t'_i einzusetzen – die in `typecase` inhärente β-Reduktion wird damit vom partiellen Auswerter vollzogen.

Es ist wichtig sich klar zu machen, daß $\mathcal{L}$ das Funktionsargument unausgewertet in t'_i einsetzt – es wird genau ein β-Reduktionsschritt vollzogen! Um doppelte Argumentauswertung zu verhindern, könnte man $\mathcal{L}$ so umschreiben, daß anstelle einer β-Reduktion ein `let`-Ausdruck generiert wird.

Primär fokussiert $\mathcal{T}$ auf die Reduktion von Termen, die von Typen abhängen, und der Elimination der β-Reduktion der `typecase`-Semantikregeln. Natürlich könnten wir $\mathcal{T}$ so erweitern, daß auch normale Terme partiell ausgewertet werden. Derartige Optimierungen sind für unser Vorhaben jedoch von sekundärer Bedeutung. Die einzige Term-Optimierung, die wir voraussetzen, ist die partielle Auswertung von Projektionen.

Durch die Beschränkung auf einen pränex-prädikativen Kalkül können wir garantieren, daß von $\mathcal{T}$ alle Typapplikationen erfaßt werden. Λ-Abstraktionen dürfen nur auf oberster Ebene, oder in `let`-Ausdrücken auftreten. `let`-Ausdrücke haben wir aber gerade so konstruiert, daß polymorphe Funktionen instanziiert werden müssen.

$\mathcal{T}$ hat interessante Eigenschaften. Zum einen ist die Transformation korrekt. Sei t ein wohlgetypter, abgeschlossener System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Term, dann gilt:

Korrektheit von $\mathcal{T}$	$\vdash t : T \;\wedge\; t \xrightarrow{*} v \Rightarrow \mathcal{T}[\![t]\!] \xrightarrow{*} v$

Den Beweis führt man durch Induktion über den Termaufbau von t, wobei man sich zu Nutze macht, daß der Term t wohlgetypt ist. Man kann also davon ausgehen, daß die Prämissen der zu einem Term t gehörenden Typregel erfüllt sind.

Es kann garantiert werden, daß $\mathcal{T}$ für jeden wohlgetypten Term terminiert:

Terminierung von $\mathcal{T}$	$\vdash t : T \;\Rightarrow \mathcal{T}[\![t]\!]$ terminiert

Den Beweis kann man ebenfalls recht einfach durch strukturelle Induktion über den Termaufbau führen. Intuitiv kann man sich diese Eigenschaft von $\mathcal{T}$ aber auch dadurch klar machen, daß die Typisierbarkeit von Termen in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ entscheidbar ist und zu jedem Typ eine Normalform gefunden werden kann. Da wir den Laufzeitteil von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ abgesehen von Ein-Schritt-β-Reduktionen unangetastet lassen, ergibt sich die Terminierung von $\mathcal{T}$.

In Einheit mit dem partiellen Auswerter $\mathcal{T}$ kann System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ zu den sogenannten **Mehrstufigen Sprachen** (engl.: *multilevel languages*) gezählt werden. In diesen Sprachen verläuft die Programmausführung in mehreren, aufeinanderfolgenden Phasen. In den ersten $n-1$ Phasen werden spezialisierte Programme generiert; erst die letzte Ausführungsphase berechnet das gewünschte Ergebnis. Die Zuordnung eines Codefragments zu einer Ausführungsphase erfolgt explizit durch spezielle Annotationen im Quelltext.

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ist eine zweistufige Sprache: Stufe 0 wird von Typausdrücken und `typecase` geformt und vom partiellen Auswerter $\mathcal{T}$ abgearbeitet und somit von der Reduktionsrelation $\rightsquigarrow$ dominiert. Stufe 1 entspricht der Reduktion von Termen durch Anwendung von $\rightarrow$.

Die Separation der beiden Stufen (engl.: *phase distinction* [66]) erfordert keine expliziten

Annotationen, sondern ergibt sich implizit aus der Trennung von Typanalyse und Programmausführung.

3.4 Einfache Anwendungen der strukturellen Typanalyse

`Typecase` und `typecase` reichen eigentlich schon aus, um die Vorgänge bei der C++ Template-Metaprogrammierung zu erklären. In diesem Abschnitt werden wir zusätzlich noch zwei Anwendungen der strukturellen Typanalyse (nebst syntaktischem Zucker) vorstellen, die uns später zu suggestiveren Schreibweisen verhelfen.

3.4.1 *Pattern Matching*

Summentypen erlauben das Zusammenfassen mehrerer Produkttypen zu einem Typ. Zum Beispiel definiert

```
data Sum = C Integer Integer | N
```

in Haskell den Typ `Sum`, dessen Werte sich aus der Vereinigung der Konstruktortypen `C Integer Integer` und `N` ergeben.

Zur sinnvollen Arbeit mit Summentypen muß man den Konstruktortyp eines Wertes zur Laufzeit abfragen können – z.B. ist die Anwendung der Projektion auf einen Wert vom Typ `N` unsinnig, für Werte vom Typ `C Integer Integer` hingegen nicht.

Die Zuordnung eines Summentypwertes zu einem Konstruktortyp wird *case*-Analyse genannt. Ein entsprechendes Haskell-Sprachkonstrukt haben wir bereits flüchtig kennengelernt – `case`-Terme bieten aber mehr als nur die reine *case*-Analyse:

```
name s = case s of
           N       -> ''N''
           (C x y) -> ''C'' ++ x ++ y
```

In der Funktion `name` wird der Wert `s` vom Typ `Sum` darauf hin untersucht, ob er mit Hilfe des Konstruktors `N`, oder aber mit Hilfe des Konstruktors `C` erzeugt wurde (*case*-Analyse). Trifft der zweite Fall zu, so werden die Variablen `x` und `y` an die zugehörigen Konstruktorkomponenten gebunden und in der rechten Seite von `->` entsprechend substituiert. Diesen Vorgang nennt man ***pattern matching***.

Auf den ersten Blick hat Haskells `case`-Kommando eine gewisse Ähnlichkeit mit `typecase`. Aber Vorsicht: Hier werden Werte und keine Typen hinsichtlich ihrer Struktur analysiert. Darüber hinaus bieten weder `typecase` noch `Typecase` die Möglichkeit zum *pattern matching* – zumindest nicht so unmittelbar.

Aufgrund der strukturellen Äquivalenz von Wert und Typ, kennen wir nach erfolgter *typecase*-Analyse die Struktur des zugehörigen Wertes und können darauf aufbauend ein *pattern matching* für diesen realiseren. Die *case*-Analyse operiert demnach auf Typen, während das *pattern matching* sich auf Werte bezieht. Der kleine aber feine Unterschied zu Haskell ist, daß wir in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ Typberechnungen via $\mathcal{T}$ vor der Laufzeit abarbeiten können. Die *case*-Analyse wird damit zu einem Compilezeit-Mechanismus.

Betrachten wir ein Beispiel: Angenommen wir wollten in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ eine Funktion bereitstellen, die ein Argument von einem Typ übernimmt, der zum Muster $\mathtt{C}(p_1, \mathtt{D}(p_2, p_3))$ paßt (p_1, p_2 und p_3 seien Platzhalter für beliebige monomorphe Typen). Sofern der Typ eines konkreten Argumentes mit diesem Muster unifiziert werden kann, hat der zugehörige Wert eine äquivalente Struktur und wir können die Variablen im Typmuster an die zugehörigen Werte binden (Anmerkung: da p_1 bis p_n an Werte gebunden werden, haben wir im Muster nicht die sonst für Monotypes übliche Metavariable M verwendet).

Folgender Pseudocode verdeutlicht unsere Idee:

$$
\begin{array}{l}
\mathtt{foo} := \Lambda\delta :: *.\mathtt{if}\ \delta\ \mathtt{unifies}\ \mathtt{C}(p_1, \mathtt{D}(p_2, p_3))\ \mathtt{then} \\
\qquad \mathtt{let}\ \sigma = \mathtt{unify}(\delta, \mathtt{C}(p_1, \mathtt{D}(p_2, p_3))) \\
\qquad\quad \mathtt{in}\ \lambda v : \mathtt{C}(\sigma(p_1), \mathtt{D}(\sigma(p_2), \sigma(p_3))).\mathtt{let}\ x_1 = v.0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad x_2 = v.1.0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad x_3 = v.1.1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{in}\ \mathtt{F}(x_1, x_2, x_3)
\end{array}
$$

Die Funktion `unifies` testet ob der Typ δ und das Muster $\mathtt{C}(p_1, \mathtt{D}(p_2, p_3))$ unifizierbar sind. Den Unifikator σ erhält man mit Hilfe der Funktion `unify`.

Eine entsprechende Konstruktion können wir mit `Typecase` und `typecase` realisieren. Die Idee ist, sowohl den Argument-, als auch den Ergebnistyp der Funktion `foo` über `Typecase` zu berechnen, so daß `foo` den Typ

$$\forall\alpha.\mathtt{Match}[\alpha] \to \mathtt{Ret}[\alpha]$$

erhält.

Die Funktion `Match` wirkt für Typen, die zum Muster $\mathtt{C}(p_1, \mathtt{D}(p_2, p_3))$ passen, wie eine Identitätsfunktion. In allen anderen Fällen liefert sie `void`:

$$
\begin{array}{l}
\mathtt{Match} := \Lambda\delta :: *.\mathtt{Typecase}\ \delta\ \mathtt{of} \\
\qquad\qquad \mathtt{C}: \quad \Lambda\alpha_0 :: *.\Lambda\alpha'_0 :: K.\Lambda\alpha_1 :: *.\lambda\alpha'_1 :: K. \\
\qquad\qquad\qquad\quad \mathtt{Typecase}\ \alpha_1\ \mathtt{of} \\
\qquad\qquad\qquad\qquad \mathtt{D}: \quad \Lambda\alpha_{10} :: *.\Lambda\alpha'_{10} :: K.\Lambda\alpha_{11} :: *.\lambda\alpha'_{11} :: K. \\
\qquad\qquad\qquad\qquad\qquad\quad \delta \\
\qquad\qquad\qquad\qquad \mathtt{default}: \quad \lambda\alpha :: *.\mathtt{void} \\
\qquad\qquad \mathtt{default}: \quad \Lambda\alpha :: *.\mathtt{void}
\end{array}
$$

In `Match` wird kein Gebrauch von der rekursiven Struktur von `Typecase` gemacht – Ergebnisse, die sich aus der *bottom-up* Berechnung ergeben, werden ignoriert. Im wesentlichen besteht die Aufgabe von `Match` darin, sicherzustellen, daß an den entsprechenden Stellen im Typargument ein zum Muster passender Konstruktor steht.

Da es in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ keine Werte vom Typ `void` gibt, kann `foo` nur aufgerufen werden, wenn der Funktionswert zum Muster paßt. Effektiv schränkt `Match` damit den Definitionsbereich von `foo` auf die für uns interessanten Fälle ein.

Eine allgemeine Vorschrift zur Bildung einer zu einem Muster passenden Funktion `Match` ist nicht schwer zu finden. Abbildung 3.9 skizziert eine entsprechende Transformation.

$$
\begin{array}{lll}
\alpha \text{ Typvariable} & C_{\texttt{Match}}[\![\alpha]\!](\alpha_u) & = \epsilon \\
\sigma_{\mathcal{C}}(\texttt{C}) = 0 & C_{\texttt{Match}}[\![\texttt{C}]\!](\alpha_u) & = \texttt{Typecase } \alpha \texttt{ of} \\
 & & \quad \texttt{default}: \ \Lambda\alpha :: *.\texttt{void} \\
 & & \quad \texttt{C}: \\
\sigma_{\mathcal{C}}(\texttt{C}) = 1 & C_{\texttt{Match}}[\![\texttt{C}(x)]\!](\alpha_u) & = \texttt{Typecase } \alpha \texttt{ of} \\
 & & \quad \texttt{default}: \ \Lambda\alpha :: *.\texttt{void} \\
 & & \quad \texttt{C}: \ \Lambda\alpha_{u\circ 0} :: *.\Lambda\alpha'_{u\circ 0} :: K.\circ \\
 & & \qquad C_{\texttt{March}}[\![x]\!](\alpha_{O0}) \\
\sigma_{\mathcal{C}}(\texttt{C}) = 2 & C_{\texttt{Match}}[\![C(x,y)]\!](\alpha_u) & = \texttt{Typecase } \alpha \texttt{ of} \\
 & & \quad \texttt{default}: \ \Lambda\alpha :: *.\texttt{void} \\
 & & \quad \texttt{C}: \ \Lambda\alpha_{u\circ 0} :: *.\Lambda\alpha'_{u\circ 0} :: K. \\
 & & \qquad \Lambda\alpha_{u\circ 1} :: *.\Lambda\alpha'_{u\circ 1} :: K.\circ \\
 & & \qquad C_{\texttt{Match}}[\![x]\!](\alpha_{o\circ 0})\circ \\
 & & \qquad C_{\texttt{Match}}[\![y]\!](\alpha_{o\circ 1}) \\
 & TC_{\texttt{Match}}[\![x]\!] & = \Lambda\delta :: *. \circ C_{\texttt{Match}}[\![x]\!](\delta_\epsilon) \circ \delta
\end{array}
$$

Abbildung 3.9: Skizze einer Funktion $TC_{\texttt{Match}}[\![x]\!]$, die ausgehend vom Muster x eine Typfunktion generiert, die für alle Typen, die nicht zum Muster x passen, `void` als Ergebnis liefert. $\circ$ beschreibt die Wortkonkatenation; ϵ steht für das leere Wort. Im Index von α generieren wir die globale Termposition des entsprechenden Unterterms und stellen so einen unmittelbaren Bezug zum Muster her.

Zur Berechnung des Ergebnistyps verfolgen wir eine ähnliche Strategie. Sei `FALSE` ein nullstelliger Konstruktor. Paßt der an `Foo` übergebene Typ nicht zum Muster, wird `FALSE` zurückgegeben; ansonsten wird der Ergebnistyp wie gewünscht berechnet:

$$
\begin{array}{l}
\texttt{Ret} := \Lambda\delta :: *.\texttt{Typecase } \delta \texttt{ of} \\
\qquad \texttt{C}: \ \Lambda\alpha_0 :: *.\Lambda\alpha'_0 :: *.\Lambda\alpha_1 :: *.\lambda\alpha'_1 :: *. \\
\qquad\quad \texttt{Typecase } \alpha_1 \texttt{ of} \\
\qquad\qquad \texttt{D}: \ \Lambda\alpha_{10} :: *.\Lambda\alpha'_{10} :: K. \\
\qquad\qquad\quad \Lambda\alpha_{11} :: *.\lambda\alpha'_{11} :: K. \\
\qquad\qquad\quad \texttt{F}(\alpha_1, \alpha_{10}, \alpha_{11}) \\
\qquad\qquad \texttt{default}: \ \Lambda\alpha :: *.\texttt{FALSE} \\
\qquad \texttt{default}: \ \Lambda\alpha :: *.\texttt{FALSE}
\end{array}
$$

Auch in diesem Fall bleiben die rekursiven Möglichkeiten von `Typecase` ungenutzt und man kann leicht eine generelle Vorschrift zur Berechnung des Rückgabewertes angeben. Einzige Voraussetzung ist, daß im Term $\texttt{F}(x,y,z)$ die Variablennamen so angepaßt werden, daß sie mit $C_{\texttt{Match}}$ kompatibel sind; d.h. in α_o umbenannt werden, wobei o der Termposition der zugehörigen Variable im Musterterm entspricht. In unserem Beispiel führt dieses Vorgehen zum Term $F(\alpha_0, \alpha_{10}, \alpha_{11})$.

Entspricht das Typargument nicht dem Muster, so hat `foo` den Typ

$$\texttt{void} \rightarrow \texttt{FALSE}$$

Da kein Wert dem Typ `void` zugeordnet ist, kann diese Funktion jedoch nie aufgerufen werden!

$$TC_{\texttt{RET}}[\![x]\!] = \Lambda\delta :: *. \circ C_{\texttt{Match}}[\![x]\!] \circ T$$

Abbildung 3.10: Skizze: Synthetisierung einer Funktion zur Berechnung des Ergebnistyps von `foo`.

Hinweis: Im Fehlerfall dürfen wir in `Ret` natürlich nicht `void` zurückgeben. Aus Sicht des Typsystems könnte sonst sehr wohl ein Wert von diesem Typ entstehen und wir müßten im Typsystem weitergehende Vorkehrungen treffen.

`foo` selbst kann nun wie folgt realisiert werden:

$$\begin{array}{l}\texttt{foo} := \Lambda\delta :: *.\texttt{typecase Match}[\delta] \quad \langle\Lambda\gamma :: *.\texttt{Match}[\gamma] \to \texttt{Foo}[\gamma]\rangle \texttt{ of} \\ \qquad\qquad \texttt{void}: \quad \lambda x : \texttt{void.FALSE} \\ \qquad\qquad \texttt{default}: \quad \Lambda\alpha :: *.\lambda z : \alpha.\texttt{let } p_0 = z.0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p_{10} = z.1.0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p_{12} = z.1.1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{in F}(p_0, p_{10}, p_{11})\end{array}$$

Paßt δ nicht zum gewünschten Muster, so könnten wir im `void`-Fall eine erneute strukturelle Typanalyse durchführen, um z.B. auf Entsprechung mit dem Muster $E(x, y)$ zu testen – die Berechnung des Rückgabetyps von `foo` müßte dazu natürlich geeignet angepaßt werden.

Allgemein können wir festhalten, daß sich mit `Typecase` und `typecase` ein gestufter *pattern matching* Mechanismus realisieren läßt. Gestuft deshalb, weil die strukturelle Analyse allein im Typsystem erfolgt (Typäquivalenz), während die Berechnung der zum Muster passenden Funktion mit Hilfe der Reduktionsrelation ($\to$) realisiert wird. Im Gegensatz dazu ist *pattern matching* in Haskell einstufig, da Konstruktorstrukturanalyse und Funktionsberechnung zur Laufzeit erfolgen.

Das *pattern matching* über drei Funktionen zu beschreiben, ist aufwendig und unübersichtlich. Wir werden daher drei neue Schreibweisen (*syntax sugar*) einführen: `Match`-, `match`- und `case`-Terme.

Mit `Match`-Termen lassen sich Typen in Abhängigkeit von einem Typmuster definieren; syntaktisch haben sie die folgende Gestalt:

$$\texttt{Match } T \texttt{ with } P \Rightarrow T_1 \texttt{ else } T_2$$

Dabei ist P ein Muster, also ein monomorpher Typ, der Typvariablen enthalten darf, T_1 und T_2 sind Typen. In T_1 dürfen die Mustervariablen aus P vorkommen, in T_2 hingegen nicht. Paßt der Typ T zum Muster P, wird der Typ T_1 zurückgegeben, wobei die Mustervariablen entsprechend ersetzt werden. Konnte keine Übereinstimmung von T und P festgestellt werden, ist das Ergebnis des `Match`-Terms der Typ T_2.

`match`-Terme sind genauso aufgebaut, liefern jedoch Terme als Ergebnis:

$$\texttt{match } T \langle F\rangle \texttt{ with } P \Rightarrow t_1 \texttt{ else } t_2$$

Die Funktion F zur Berechnung des Ergebnistyps lassen wir manchmal auch weg. Da wir von der rekursiven Struktur von `typecase` keinen Gebrauch machen, lassen sich die Ergebnistypen unmittelbar aus den jeweiligen Behandlungsfällen inferieren.

Sofern für uns nicht weiter interessant, werden wir auch den `else`-Zweig in `Match` und `match`-Anweisungen auslassen und um tief geschachtelte Terme zu vermeiden, führen wir in Anlehnung an `let`-Ausdrücke die folgende Kurschreibweise ein:

$$\begin{array}{lll} \texttt{match}\ T\ \texttt{with} & P_1 & \Rightarrow\ t_1 \\ & \vdots & \\ & P_n & \Rightarrow\ t_n \\ & \texttt{else} & t_e \end{array}$$

Wenn wir ein Funktionsmakro definieren, erlauben wir in den Ergebnistermen auch rekursive Aufrufe, aber nur, wenn sie mit dem Rekursionsmuster von `typecase` in Einklang stehen; z.B.

$$\begin{array}{lll} \texttt{dynLength} := \Lambda\delta :: *.\texttt{match}\ \delta\ \texttt{with} & \texttt{N} & \Rightarrow\ \lambda n : N.0 \\ & \texttt{C}(\alpha) & \Rightarrow\ \lambda x : C(\alpha).1 + \texttt{dynLength}[\alpha]\ x.0 \end{array}$$

Als letzte Ergänzung führen wie schließlich `case`-Terme ein, um nach einem erfolgreichem matching komfortabler auf die Komponenten eines Wertes zugreifen zu können (dazu hatten wir im obigen Beispiel `let`-Expressions eingesetzt).

$$\texttt{case}\ x : T\ \texttt{of}\ \langle P_t\ ;\ P_v\rangle \Rightarrow t_1\ \texttt{else}\ t_2$$

Im Vergleich zu `match` und `Match` operieren `case`-Expressions auf einem Paar aus Wert und Typ und ein Muster besteht aus zwei Komponenten: Dem Typmuster P_t und einem Wertmuster P_v.

Ein Wertmuster ist entweder eine Termvariable, oder aber ein Konstruktorterm, der Termvariablen enthalten darf. In jedem Fall muß das Wertmuster bezüglich der Matchordnung (siehe Seite 27) kleiner sein als das Typmuster.

Betrachten wir ein Beispiel. Angenommen man wolle Werte, die zum Typmuster $\texttt{C}(\alpha, \texttt{D}(\beta,\gamma))$ passen, in Werte vom Typ $\texttt{F}(\alpha, \texttt{D}(\beta,\gamma))$ wandeln, also den Konstruktor C durch den Konstruktor F ersetzen. Dann kann man dazu folgenden `case`-Term verwenden:

$$\begin{array}{lll} \texttt{bar} := \Lambda\delta :: *.\lambda d : \delta.\texttt{case}\ d : \delta\ \texttt{of}\ \langle \texttt{C}(\alpha, \texttt{D}(\beta,\gamma))\ ;\ \texttt{C}(x,y)\rangle & \Rightarrow & \texttt{F}(x,y) \\ \texttt{else} & & \texttt{FALSE} \end{array}$$

Das Termmuster ist in diesem Beispiel genereller als das Typmuster, da wir auf die Komponenten des Konstruktors D nicht zugreifen müssen. Die Variable y wird folglich an einen Wert vom Typ $\texttt{D}(\beta,\gamma)$ gebunden.

Auch wenn der Name es suggeriert, haben System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-`case`-Terme nichts mit Haskells `case`-Expressions zu tun. Das *pattern matching* bezieht sich auf Typnamen und wird zur Übersetzungszeit vollzogen.

Wir werden `match`, `Match` und `case` erst später einsetzen, wenn wir Parallelen zwischen C++ und System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ziehen werden.

3.4.2 Überladung

Eine interessante Anwendung der strukturellen Typanalyse ist die Überladung von Bezeichnern (auch *ad-hoc*-Polymorphie genannt - siehe [30]).

Seien $==_{\texttt{int}}$ und $==_{\texttt{bool}}$ vordefinierte Funktionen für den Vergleich von `int`- und `bool`-Werten. Wir wollen jetzt eine Funktion `eq` bereitstellen, mit der wir zwei `int`-Werte, zwei boolesche Werte und solche Werte, deren Typ aus der Kombination der Konstruktoren $\times$, `int` und `bool` entstanden ist, vergleichen können.

Da aus der Kombination von $\times$, `int` und `bool` unendlich viele Typen hervorgehen können, müßten wir potentiell unendlich viele Überladungen von `eq` implementieren. Wie wir gleich sehen werden, kann man sich auf die tatsächlich benötigten Überladungen beschränken und diese automatisch über `typecase` generieren lassen.

Zunächst stellt man, ähnlich wie zur Konstruktion von `match`, eine Typfunktion zur Verfügung, mir der der Definitionsbereich von `eq` auf die relevanten Fälle eingeschränkt wird. Eine Beschränkung ist notwendig, da z.B. der Vergleich von Funktionen nicht entscheidbar ist:

$$
\begin{array}{l}
\texttt{Eq} := \Lambda\delta :: *.\texttt{Typecase}\ \delta\ \texttt{of} \\
\qquad\begin{array}{ll}
\texttt{int}: & \texttt{int} \\
\texttt{bool}: & \texttt{bool} \\
\times: & \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K. \\
 & \times(\alpha', \beta') \\
\texttt{default}: & \Lambda\alpha :: *.\texttt{void}
\end{array}
\end{array}
$$

Zur Erinnerung: α' und β' resultieren aus der rekursiven Anwendung von `Eq` und es gilt: $\texttt{Eq}[\alpha] = \alpha'$ bzw. $\texttt{Eq}[\beta] = \beta'$.

Die Wertfunktion `eq` analysiert einen Typ δ und generiert eine geeignete Vergleichsfunktion:

$$
\begin{array}{l}
\texttt{eq} := \Lambda\delta :: *.\texttt{typecase}\ \delta\ \ \langle\Lambda\gamma :: *.\texttt{Eq}[\gamma] \to \texttt{Eq}[\gamma] \to \texttt{bool}\rangle\ \texttt{of} \\
\qquad\begin{array}{ll}
\texttt{int}: & ==_{\texttt{int}} \\
\texttt{bool}: & ==_{\texttt{bool}} \\
\times: & \Lambda\alpha :: *.\lambda f_0 : Eq[\alpha] \to Eq[\alpha] \to \texttt{bool}. \\
 & \Lambda\beta :: *.\lambda f_1 : Eq[\beta] \to Eq[\beta] \to \texttt{bool}. \\
 & \lambda x : Eq[\times(\alpha, \beta)].\lambda y : Eq[\times(\alpha, \beta)]. \\
 & f_0(x.0)(y.0)\ \texttt{and}\ f_1(x.1)(y.1) \\
\texttt{default}: & \Lambda\alpha :: *.\lambda x : \texttt{void}.\lambda y : \texttt{void}.\texttt{false}
\end{array}
\end{array}
$$

Der Vergleich von Tupeln vom Typ $\times(\alpha, \beta)$ erfolgt komponentenweise unter Rückführung auf die für die Typen α und β generierten Vergleichsfunktionen f_0 und f_1.

Bevor die Funktion `eq` auf zwei Werte angewendet werden kann, muß sie mit dem Typ der zu vergleichenden Argumente versorgt werden – dabei wird die eigentliche Vergleichsfunktion generiert.

Da die Auswahl einer überladenen Funktion grundsätzlich in Abhängigkeit der Argumenttypen erfolgt, erscheint diese Term-Typ-Applikation redundant – sie könnte unmittelbar aus den Argumenttypen inferiert werden.

Wir vereinbaren daher $f\,[\![a]\!]$ als Kurzschreibweise für $f[T](a)$, wobei T der Typ von a ist. Diese spezielle Form der Funktionsapplikation nennen wir **Überladungsapplikation**.

Zur Integration in System $\mathbf{F}_\omega$ ergänzen wir Typ- und Semantikregel um:

$\boxed{\Gamma \vdash t : T}$ System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ + Überladung

$$\frac{\Gamma \vdash f : \forall\alpha :: K.\alpha \to T_2 \quad \Gamma \vdash a : T_1 \quad \Gamma \vdash f[T_1] \vdash T_1 \to T_2}{\Gamma \vdash f\,[\![a]\!] : T_2} \textbf{ (T-OvlApp)}$$

$\boxed{t_1 \to t_2}$

$$\frac{\vdash a : T}{(\Lambda\alpha :: *.\lambda x : \alpha.t)\,[\![a]\!] \to (\Lambda\alpha :: *.\lambda x : \alpha.t)[T](a)} \textbf{ (E-OvlApp)}$$

Typen zur Laufzeit zu inferieren wäre absolut ungeschickt und würde unserem Prinzip der strikten Trennung von Wert- und Typberechnungen widersprechen. Wir erweitern daher den partiellen Auswerter $\mathcal{T}$, so daß er die Typinferenz und Typapplikation vollzieht:

$\mathcal{T}$

$$\mathcal{T}[\![t\,[\![t_1]\!]]\!] = \mathcal{L}[\![\mathcal{T}[\![t[T]]\!](\mathcal{T}[\![t_1]\!])]\!] \quad \text{falls } \Gamma \vdash t_1 : T$$

Das Typsystem garantiert, daß t im Term $t\,[\![t_1]\!]$ auf einen Typ und die dabei entstehende Funktion auf einen Wert dieses Typs anwendbar ist.

Die Typapplikation kann daher durch Auswerten von $\mathcal{T}[\![t[T]]\!]$ aufgelöst werden, wobei ein Term t' entsteht. Anschließend wird das Funktionsargument t_1 zu t_1' reduziert – in t_1 könnten ja `typecase`-Terme enthalten sein. Zum Abschluß vollzieht der partielle Auswerter einen β-Reduktionsschritt (Funktion $\mathcal{L}$ - Applikation von t' auf t_1').

Um den Typ von t_1 bestimmen zu können, müßten wir $\mathcal{T}$ eigentlich eine Typisierungsumgebung Γ mitgeben und diese beim Traversieren des abstrakten Syntaxbaumes entsprechend anpassen. Da wir Γ jedoch nur für den Fall der Überladungsapplikation brauchen, setzen wir Γ als gegeben voraus.

In einer konkreten Implementierung könnte $\mathcal{T}$ einen mit Typannotationen versehenen Baum traversieren. Da $\mathcal{T}$ typerhaltend ist, neue Terme nur aus bestehenden zusammensetzt und Terme typerhaltend reduziert, kann man $\mathcal{T}$ leicht so realisieren, daß die Typannotationen im Baum erhalten bleiben.

Kehren wir noch einmal zurück zu `eq`: So, wie wir `eq` und `Eq` vorgestellt haben, können wir anstelle von $\texttt{eq}[T](a)(b)$ jetzt kurz $\texttt{eq}\,[\![a]\!]\,b$ schreiben – das Mischen von Applikation und Überladungsapplikation scheint in diesem Fall eher verwirrend.

Natürlich können wir `eq` aber auch so umschreiben, daß diese Funktion ausschließlich auf Tupeln operiert und die Tupelkomponenten miteinander verglichen werden. Dann wird der suggestivere Aufruf $\texttt{eq}\,[\![a, b]\!]$ möglich (die Tupelklammern lassen wir in diesem Fall weg). Alternativ könnte man auch auf der bestehenden Funktion `eq` aufsetzen und folgende Hilfsfunktion zur Verfügung stellen:

$$\begin{aligned}
\texttt{eq}_P := \Lambda\delta :: *.\texttt{typecase}\ \delta\ \ \langle\Lambda\gamma :: *.\gamma \to \texttt{bool}\rangle\ \texttt{of}\\
\texttt{default}:\ \ \Lambda\alpha :: *.\lambda x : \alpha.\texttt{false}\\
\times :\qquad\ \ \Lambda\alpha :: *.\lambda f_0 : \alpha \to \texttt{bool}.\\
\Lambda\beta :: *.\lambda f_1 : \beta \to \texttt{bool}.\\
\lambda c : \times(\alpha, \beta).\texttt{eq}[\alpha]\ \ c.0\ \ c.1
\end{aligned}$$

Wie man leicht nachvollzieht (siehe Anhang A.1) gilt

$$\mathcal{T}[\![\texttt{eq}_P\,[\![(7,9)]\!]]\!] \;=\; ==_{\texttt{int}} \;\times(7,9).0 \;\times(7,9).1$$

Die Wahl der „richtigen" Vergleichsfunktion erfolgt in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ zur Übersetzungszeit (man spricht auch von einem Compilezeit-Mechanismus).

Vergleicht man mit Haskell, so entsprechen `Eq` und `eq` der Typklasse `Eq`. Bei der Definition des Typs der Vergleichsfunktion ist man in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ jedoch ein wenig freier. In Standard-Haskell [136, 74] muß eine Vergleichsfunktion für den Typ α vom Typ $\alpha \to \alpha \to$ `bool` sein, da Typklassen nur von einen Typparameter abhängig sein dürfen. In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ besteht diese Beschränkung nicht.

Selbst mit neueren Erweiterungen, wie Multiparameter-Typklassen [137] oder Klassen mit funktionalen Abhängigkeiten (engl. *functional dependencies*) [83] kommt Haskell an die Flexibilität von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ nicht heran. Nehmen wir zum Beispiel an, man wolle eine Funktion `add` wie folgt überladen:

1. `add :: (Float,Int) -> Float`
2. `add :: (Int,Float) -> Int`
3. Für alle anderen Typen T: `add :: (T,T) -> T`

Die ersten beiden Fälle bekommt man in einer erweiterten Version von Haskell mit Multiparameter Typklassen und *functional dependencies* in den Griff, indem man z.B. folgende Typklasse vereinbart:

```
class Plus a b c | a b -> c
  add :: a -> b -> c
```

Der Term `a b -> c` bringt zum Ausdruck, daß der Parameter `c` abhängig von den Parametern `a` und `b` ist und dient vornehmlich zur Unterstützung bei der Typinferenz. Die ersten beiden Überladungen kann man durch folgende Instanzen implementieren:

```
instance Plus Float Int Float ...
instance Plus Int Float int ...
```

Den letzten Fall kann man im erweiterten Haskell nicht ausdrücken. In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ bereitet er keine Probleme und kann leicht unter Einsatz eines `default`-case, oder mit folgendem `match`-Term beschrieben werden:

$$\begin{array}{lll} \texttt{add} := \Lambda\alpha :: *.\texttt{match}\ \alpha & & \\ \qquad\qquad\qquad\qquad \texttt{in}\ \ \texttt{float} \times \texttt{float} & \Rightarrow & \ldots \\ \qquad\qquad\qquad\qquad\quad\ \ \texttt{int} \times \texttt{int} & \Rightarrow & \ldots \\ \qquad\qquad\qquad\qquad\quad\ \ \beta \times \beta & \Rightarrow & \ldots \end{array}$$

In Haskell wird Überladung zur Laufzeit aufgelöst [10] (man spricht von einem Laufzeit-Mechanismus). Das hat zwar negative Auswirkungen auf die Ausführungsgeschwindigkeit eines Programmes, bringt aber gewisse Vorteile bei der separaten Übersetzung von Modulen.

Beim Vergleich zwischen Haskell und System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ dürfen wir nicht außer acht lassen, daß Typen in Haskell inferiert werden, der Programmierer also auf Typannotationen im Quelltext verzichten kann. Die meisten Einschränkungen gegenüber System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ wurden im Sinne eines entscheidbaren Typinferenzverfahrens vorgenommen – der Vergleich ist also nicht ganz fair.

Wir werden auf Überladung später, am Ende des nächsten Kapitels, noch einmal zurückkommen, wenn wir die Integration von Sprachen in Haskell mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ vergleichen.

3.5 Weiterführende Literatur

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ist eine ausdrucksstarke Modellsprache. Neben den hier neu vorgestellten Anwendungen des *pattern matching*, sind `Typecase` und `typecase` erfolgreich zur Realisierung von Überladung [118], parametrischer Polymorphie [67] und zur Optimierung von *garbage collection* [118] eingesetzt worden.

Will man die separate Übersetzung von Modulen ermöglichen, die Gebrauch von struktureller Typanalyse machen, kommt man nicht umhin, Typinformation zur Laufzeit sichtbar zu machen. Stephanie Weirich diskutiert in [39] und [177] ein Verfahren, welches entsprechende Unterstützung liefert. Ihre Idee ist, Typnamen zur Laufzeit durch Terme darzustellen, wobei jedem Term exakt ein Typ zugeordnet ist (*singleton types*).

Ein mit struktureller Typanalyse vergleichbarer Ansatz, sind *polymorphic variants*, die von Garrigue in [53, 54] vorgestellt werden. *Polymorphic variants* entsprechen im wesentlichen den hier vorgestellten Konstruktortypen. Funktionen können auf bestimmte *polymorphic variants* eingeschränkt werden. Den Typ des tatsächlich an eine solche Funktion übergebenen Arguments kann man durch einen speziellen *pattern matching*-Mechanismus erfragen, so daß man auch hier Terme in Abhängigkeit von Typen formulieren kann. Ein Äquivalent zu `Typecase` gibt es allerdings nicht.

Alternativen zur Integration von Überladung in den λ-Kalkül findet man z.B. bei Castagna, Ghelli und Longo in [32]. Erweiterungen des λ-Kalküls um *pattern matching* stellen z.B. Wadler und Peyton Jones in [139, Seite 51 ff.] oder Kahl in [90] vor.

Eine der ersten Multi-level Sprachen, λ_{Mix}, wurde von Jones, Gomard und Sestoft entworfen [88, Seite 166 ff.], um partielle Auswertung von λ-Ausdrücken zu beschreiben. λ_{Mix} definiert zwei Ausführungsebenen. Die erste Ebene wird zur Zeit der partiellen Auswertung abgearbeitet und hat ein Residuum als Ergebnis, das in der sich anschließenden zweiten Phase ausgeführt wird. Eine weitere zweistufige Variante des λ-Kalküls findet man bei Nielson und Nielson [123].

Kapitel 4

Spracheinbettung mit System $F^{SA}_{\omega,1}$

Der wahrscheinlich einfachste Weg vom Programmtext zu einem ausführbaren Programm führt über die Implementierung eines Interpreters. Seine Implementierung fällt besonders leicht, wenn die Semantik einer Sprache operationell beschrieben wurde, und das Regelwerk rein syntaxgesteuert ist.

Durch die enge Verzahnung von Analyse- und Auswertungsphase ist der Interpreter-Ansatz jedoch mit einigen Nachteilen behaftet. Zum einen ist die Ausführungszeit im Vergleich zu Programmcode, der in Maschinensprache übersetzt wurde, in der Regel höher. Schwerwiegend ist aber auch, daß sämtliche Programmfehler erst zur Laufzeit erkannt werden. Schwerwiegend besonders deshalb, weil Fehlerquellen, die sich in nicht ausgeführten Programmfragmenten befinden, erst gar nicht erkannt werden können (ein *boundary-interior* Pfadüberdeckungstest, auch C2b-Test genannt, ist zwingend erforderlich).

Ein Übersetzer (engl. *compiler*), der den Quelltext zunächst in Maschinensprache übersetzt hat Vorteile. Die meisten Übersetzer sind mehrphasig aufgebaut. An die lexikalische und syntaktische Analyse schließt sich in der Regel eine semantische Analysephase an. In dieser Phase wird untersucht. ob ein Programm bestimmte Fehler verursachen kann. Im Gegensatz zu einem Interpreter ist man dabei nicht abhängig vom tatsächlichen Kontrollfluß des Programms. Vielmehr wird versucht, gewisse Programmeigenschaften durch statische Analyse zu beweisen. Zum Beispiel kann man die Herleitung eines Typs für einen Term als Beweis für das nicht-Auftreten von Typfehlern zur Laufzeit verstehen. Neben dem frühen Erkennen von Fehlern, bieten in Maschinencode übersetzte Programme natürlich klare Vorteile bezüglich der Ausführungszeit.

Leider ist die Implementierung eines Compilers erheblich komplexer, als die eines Interpreters. Zwar gibt es gute Werkzeugunterstützung zur Implementierung der syntaktischen und der semantischen Programmanalyse (z.B. lex und yacc [102]), bei der Codegenerierung liegt es jedoch allein in der Verantwortung des Programmierers, eine Übersetzungsfunktion zu implementieren, die die Semantik eines Programms nicht verfälscht. Besonders aufwendig gestaltet sich die Implementierung einer Optimierungsphase. Zum einen erfordert sie eine gewisse Sorgfalt, nur solche Optimierungen oder Kombinationen von Optimierungen zu erlauben, die die Bedeutung eines Programms nicht verändern, zum anderen verlangt sie nach einer guten Kenntnis der Spezifika der Zielmaschine.

In diesem Kapitel werden wir zeigen, wie man aus einem Interpreter durch partielle Auswertung einen Compiler gewinnen kann. Der große Vorteil wird sein, daß wir dabei einen bestehen-

den Compiler wiederverwenden können, uns also um die Implementierung der Codegenerierungs- und Optimierungsphase keine Gedanken machen müssen. Anhand einer sehr einfachen Sprache werden wir zunächst zeigen, wie man einen Interpreter in Haskell realisiert und wie daraus durch partielle Auswertung ein Übersetzer in Maschinencode entstehen könnte. Anschließend werden wir am Beispiel von MetaOCaml demonstrieren, wie man die partielle Auswertung eines Interpreters durch Einsatz von Metaprogrammierung realisiert.

Wir werden ferner zeigen, daß System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ unter ganz bestimmten Voraussetzungen ähnliche Eigenschaften mitbringt wie MetaOCaml – insbesondere kann in vielen Fällen garantiert werden, daß die Ausführungszeit eines zu interpretierenden Programmes sehr dicht an der Qualität von übersetztem Programmcode liegt.

Nachdem wir die verschiedenen Implementierungen des Interpreters vorgestellt haben, werden wir sie miteinander vergleichen und kurz darauf eingehen, wie man einen System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Übersetzer um benutzerdefinierte, domänenspezifische Transformationen erweitern kann.

In den sich anschließenden Abschnitten werden wir auf die Grenzen der Sprachintegration mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ eingehen und darstellen, wie man System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ erweitern muß, um auch komplexere Sprachen (mit Rekursion bzw. mit komplexen Typsystemen) einbetten zu können.

4.1 Vom Interpreter zum Compiler: Futamuras Projektionen

Ganz allgemein betrachtet ist ein Interpreter `inter` für eine Sprache **L** ein in einer Sprache **S** geschriebenes Programm, das ein **L**-Programm *source* und Argumente an *source* in v_d übernimmt, und daraus eine Ausgabe *out* berechnet:

$$[\![source]\!]_{\mathbf{L}}\ \ v_d = [\![inter]\!]_{\mathbf{S}}\ \ source\ \ v_d = out$$

Angenommen es stünde ein partieller Auswerter *spec* für die Sprache **S** zur Verfügung. Dann kann man aus dem **L**-Programm *source* durch Anwendung von *spec* ein **S**-Programm *target* gewinnen:

$$target = [\![spec]\!]_{\mathbf{S}}\ \texttt{inter}\ source \tag{4.1}$$

Diese Gleichung ist als **erste Futamura Projektion** bekannt und geht zurück auf Yoshihiko Futamura, der sie 1971 in [51] erstmals veröffentlichte.

Aus der Definition eines partiellen Auswerters (siehe Seite 39) ergibt sich, daß

$$[\![target]\!]_{\mathbf{S}}\ \ v = [\![\texttt{inter}]\!]_{\mathbf{S}}\ \ source\ \ v$$

gilt. Das heißt, beide Programme reagieren auf identische Eingaben mit identischen Ausgaben und sind somit äquivalent.

Wendet man den partiellen Auswerter *spec* auf sich selbst an, wobei der Interpreter *inter* als statische Eingabe verwendet wird, gewinnt man einen Übersetzer von **L** nach **S**:

$$\texttt{compiler} = [\![spec]\!]_{\mathbf{S}}(spec, \texttt{inter}) \tag{4.2}$$

Diese Gleichung heißt **zweite Futamura Projektion** und aus der Definition eines partiellen Auswerters folgt sofort die Korrektheit des Übersetzers:

$$[\![\texttt{compiler}]\!]_{\mathbf{S}}(source) = target$$

Für unser Vorhaben ist insbesondere die erste Futamura Projektion interessant. Setzt man nämlich einen Übersetzer h voraus, der **S**-Programme in die Zielsprache **T** übersetzt, so kann man einen Übersetzer von **L** nach **T** wie folgt erhalten:

$$[\![h]\!]_{\mathbf{T}}\,([\![spec]\!]_{\mathbf{S}}(\texttt{inter}, source))$$

Die Wirkung der ersten Futamura-Projektion werden wir im nächsten Abschnitt an Beispielen demonstrieren.

4.2 Vom Interpreter zum *staged interpreter*

Sprachen mit algebraischen Datentypen eignen sich hervorragend zur Implementierung von Interpretern. Besonders dann, wenn man den Interpreter zweiphasig auslegt: In der ersten Phase wird ein Programm in konkreter Syntax in eine abstrakte Notation überführt, die man mit Hilfe von algebraischen Datentypen realisiert. In der zweiten Phase läuft der eigentliche Interpretationsprozeß ab. Bei dessen Implementierung kommt einem die Darstellung des AST als Summentyp zu Gute, weil sich eine syntaxgesteuerte Auswertung sehr elegant durch *pattern matching* implementieren läßt[1].

In diesem Abschnitt werden wir eine sehr einfache Modellsprache vorstellen und zeigen, wie man Interpreter für diese in Haskell, MetaOCaml und System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ realisieren kann.

4.2.1 V: Eine einfache Modellsprache

Als Modellsprache betrachten wir einfache Terme über den ganzen Zahlen:

V

V ::=		**V-Terme**
	x	Variable
	N	Ganze Zahl
	add(V,V)	Addition
	mul(V,V)	Multiplikation

Wir verzichten auf eine formale Beschreibung der Semantik von **V**. Sie ergibt sich intuitiv aus den Rechenregeln für ganze Zahlen. Zum Beispiel ist $\texttt{add}(1, \texttt{mul}(3, 4)) = 13$.

[1] Streng genommen wird eine konkrete Sprache hier zunächst übersetzt. Der diskutierte Ansatz funktioniert grundsätzlich aber auch für „echte" Interpreter, bei denen syntaktische Analyse, semantische Analyse und Auswertung verzahnt ablaufen. Allerdings würden diese „reinen" Interpreter zu einer komplexeren Darstellung führen.

4.2.2 Ein Interpreter für V in Haskell

Die abstrakte Syntax von **V** stellen wir mit Hilfe eines Summentyps `V` dar, der für jede syntaktische Form einen Konstruktor bereithält (siehe Abbildung 4.1).

Die Hilfsfunktionen `emptyEnv` und `addEnv` dienen der einfachen Konstruktion von Umgebungen, also Zuordnungen von **V**-Variablennamen zu `Integer`-Werten.

Der Interpreter `vInter` übernimmt einen **V**-Term (einen Wert vom Typ `V`) und eine Umgebung als Argument und reduziert den **V**-Term zu einem `Integer`-Wert.

Ausgelassen haben wir den Quelltext zur Funktion `parse :: String -> V`, die aus einem **V**-Term als Zeichenkette einen Wert vom Typ `V` erzeugt.

```
data V = Var String      - Variable
       | Num Integer     - Ganze Zahl
       | Add V V         - Addition
       | Mul V V         - Subtraktion

emptyEnv            = \x -> error ''Variable nicht in Umgebung''
addEnv name val env = \x -> if (x == name) then val else (env name)

vInter (Var name) env    = env name
vInter (Num x) env       = x
vInter (Add v1 v2) env   = (vInter v1 env) + (vInter v2 env)
vInter (Mul v1 v2) env   = (vInter v1 env) * (vInter v2 env)

runVProg x y = vInter (parse ''add(x,mult(x,y))'')
                      (addEnv ''y'' y (addEnv ''x'' x emptyEnv))
```

Abbildung 4.1: Interpreter für **V** in Haskell.

Die Funktion `runVProg` veranschaulicht die Anwendung des Interpreters auf einen einfachen Term: Der Parser generiert aus dem als Zeichenkette vorliegenden **V**-Term einen AST. Der leeren Umgebung werden durch Aufruf der Funktion `addEnv` Werte für die **V**-Variablen `x` und `y` hinzugefügt und anschließend der Interpreter gestartet.

`vInter` berechnet den Wert eines Terms induktiv aus den Werten der einzelnen Teilterme. Dazu wird der AST einer `case`-Analyse unterzogen. Im Falle einer Variablen, wird deren Wert in der Umgebung nachgeschlagen; der Wert von numerischen Konstanten ergibt sich durch Extraktion aus dem umschließenden Konstruktor. Zur Auswertung von `Add`- und `Mul`-Knoten werden zunächst die Werte der beiden Operanden bestimmt. Dazu wird der Interpreter rekursiv auf die Konstruktorkomponenten des Knotens angewendet, um die dabei entstehenden Werte anschließend mit der zugehörigen Operation zu verknüpfen.

Betrachten wir `runVProg` unter dem Gesichtspunkt der partiellen Auswertung. Der Funktion `parse` stehen zur Übersetzungszeit alle Argumente zur Verfügung – der Funktionsaufruf kann vollständig ausgewertet werden, so daß das Parsing zur Übersetzungszeit durchgeführt wird und der Ausdruck `parse "add(x,mul(x,y))"` durch den Wert

```
Add (Var "x") (Mul (Var "x") (Var "y"))
```

ersetzt werden kann.

Das erste Argument an `vInter` wird dadurch statisch, das zweite (die Umgebung) ist jedoch dynamisch, da die Werte von `x` und `y` erst zur Laufzeit bekannt sind. Der Aufruf des Interpreters kann deshalb nur partiell ausgewertet werden:

```
vInter (Add (Var "x") (Mul (Var "x") (Var "y"))) e
=  (vInter (Var "x") e)  + (vInter (Mul ((Var "x") (Var "y"))) e)
= e "x" +  (vInter (Mul (Var "x") (Var "y")) e)
= e "x" + (  (vInter (Var "x") e)  * (vInter (Var "y") e) )
= e "x" + ( e "x" * +  (vInter (Var "y") e)  )
= e "x" + ( e "x" * e "y")
```

Die Zugriffe auf die Umgebung könnten ebenfalls partiell ausgewertet werden, so daß am Ende der Term `x + (x*y)` als Residuum verbleibt.

Was ist passiert? Der partielle Auswerter hat das Traversieren des AST erledigt – er hat die Struktur des Terms analysiert und damit die *case*-Analyse in die Übersetzungszeit verlagert. Darüber hinaus hat er abhängig von der Programmstruktur durch *function unfolding* Haskell-Code zur Berechnung des **V**-Terms generiert.

Der Effekt der partiellen Auswertung entspricht damit der Übersetzung eines **V**-Terms nach Haskell, wobei die Korrektheit der Übersetzung garantiert ist (erste Futamura-Projektion!). Die Umsetzung in Maschinensprache wird anschließend vom Haskell-Compiler übernommen, so daß man im Ergebnis einen Übersetzer von **V** nach Maschinensprache erhält.

Leider garantiert der Haskell Standard nicht, daß die soeben skizzierte partielle Auswertung erfolgt. Man kann also nicht mit Gewißheit voraussagen, ob mit der Implementierung des Interpreters unter dem Strich die Laufzeitqualitäten eines Compilers erzielt werden können.

Im nächsten Abschnitt stellen wir mit MetaOCaml eine Programmiersprache vor, in der man die partielle Auswertung des Interpreters explizit programmieren kann.

4.2.3 Ein *staged interpreter* für V in MetaOCaml

MetaOCaml [112] ist eine Erweiterung der funktionalen Programmiersprache OCaml um Sprachkonstrukte zur Unterstützung von Metaprogrammierung. Metaprogrammierung beschreibt das Erstellen von Programmen, die andere Programme (auch sich selbst) manipulieren. Das manipulierende Programm nennt man **Metaprogramm**, das manipulierte Programm heißt **Objektprogramm**. Einfache Beispiele für Metaprogramme sind Übersetzer und Interpreter.

Die Syntax von MetaOCaml bietet drei spezielle Konstrukte zum Generieren, Kombinieren und Ausführen von Objektprogrammen. Die Programmausführung verläuft wie bei Multi-Level-Sprachen in mehreren Phasen, allerdings ist deren Abfolge nicht durch die Sprache fest vorgegeben, sondern unterliegt der alleinigen Kontrolle des Programmierers. Zur Abgrenzung von Multi-Level-Sprachen spricht man bei MetaOCaml von einer **Multi-Stage**-Sprache [159].

Abbildung 4.2 zeigt die Implementierung des **V**-Interpreters in MetaOCaml, wobei wir die Implementierung der Funktionen `emptyEnv`, `addEnv` und `parse` ausgespart haben.

```
type v = Num of int
       | Add of v * v
       | Mult of v * v
       | Var of string

let rec vInter b env =
match b with
  Num n     -> .< n >.
| Var s     -> .< .~env s .>
| Add e1 e2 -> .< .~(vInter e1 args) + .~(vInter e2 args) >.
| Mul e1 e2 -> .< .~(vInter e1 args) + .~(vInter e2 args) >.

let runVProg x y = vInter (parse "add(x,mult(x,y))")
                          (addEnv "x" x (addEnv "y" y emptyEnv))
```

Abbildung 4.2: Ein einfacher *staged interpreter* für **V** in MetaOCaml.

Wie in Haskell, stellen wir den AST von **V**-Termen durch einen Summentyp dar. Die in der Funktion `vInter` verwendete `match`-Anweisung ist das von ML geerbte Gegenstück zu Haskells `case`-Ausdrücken und dient der *case analysis*. Größere syntaktische Unterschiede zu Haskell ergeben sich durch die zahlreichen Quotierungen, deren Bedeutung wir nun enträtseln werden.

Unquotierter Programmcode wird in Phase 0 ausgeführt. Die Auswertung von Programmcode, der von dem Klammerpaar `.<` und `>.` umschlossen ist, wird zurückgestellt. Das Ergebnis von quotiertem Code ist kein Wert, sondern Objektcode, der zu einem späteren Zeitpunkt durch Anwenden des *run*-Operators `!.` ausgeführt werden kann.

Innerhalb von quotiertem Code kann die Auswertung von Teilausdrücken durch Einsatz des *escape*-Operators `.~` erzwungen werden. Das Ergebnis ist dann allerding kein Wert, sondern Objektcode, dessen Ausführung unmittelbar zu diesem Wert führt.

Im Interpreter `vInter` wurde der Programmcode in allen Behandlungsfällen quotiert. Seine Ausführung führt daher nicht zu einem `int`-Wert, sondern zu Objektcode der die Auswertung eines **V**-Terms zu einem `int`-Wert vollzieht.

Zu beachten ist, daß aus Gründen der Typkonsistenz auch im Fall von numerischen Konstanten (`Num`-*case*) Objektcode generiert werden muß. Der Wert einer Variablen kann zur Ausführungszeit des Interpreters aus der Umgebung ermittelt werden – ohne Einsatz des *escape*-Operators würde das Nachschlagen in der Umgebung in die Laufzeit des generierten Objektcodes verlagert.

Die Kombination von Objektcodefragmenten wird bei der Behandlung von `Add`- und `Mul`-Knoten besonders deutlich. Der *escape*-Operator sorgt dafür, daß Programmcode zur Berechnung der Operanden zur Laufzeit des Interpreters generiert wird. Die aus den rekursiven Aufrufen gewonnenen Objektcodefragmente werden anschließend so verknüpft, daß ein Codefragment entsteht, welches die Addition bzw. die Multiplikation vollzieht.

`runVProg` liefert hier also keinen Wert, sondern Objektcode, wie der folgende Mitschnitt einer MetaOCaml Intepretersitzung zeigt:

```
# let a = runVProg 2 3

val a : int code = .<(2 + (2 * 3))>.
```

Als Resultat erhalten wir den MetaOCaml Term `.<(2 + (2*3))>.`, also Objektcode, dessen Auswertung `8` ergibt. Um vergleichbar mit Haskell zu sein, müssen wir die Implementierung von `runVProg` daher um die Anwendung des *run*-Operators erweitern:

```
let runVProg x y = .! (vInter (parse "add(x,mult(x,y))")
                              (addEnv "x" x (addEnv "y" y emptyEnv)) )
```

Im Ergebnis erhalten wir einen Interpreter, der in zwei explizit voneinander getrennten Phasen ausgeführt wird – man spricht von einem ***staged interpreter*** [161, 114]. Der Effekt ist derselbe, als wenn man den Haskell-Interpreter partiell auswerten würde. In MetaOCaml ist die partielle Auswertung jedoch explizit steuerbar und somit garantiert.

4.2.4 Ein *staged interpreter* für V in System $\mathbf{F}^{\mathbf{SA}}_{\omega,1}$

Wir wollen jetzt zeigen, daß sich das Konzept des *staged interpreter* unter bestimmten Voraussetzungen auch in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ realisieren läßt. Wir setzen voraus, daß System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ um einen Datentyp `string` zur Darstellung von Zeichenketten und geeignete Routinen zur `string`-Verarbeitung erweitert wurde.

Den abstrakten Syntaxbaum zu einem **V**-Term stellen wir in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ über mehrere Konstruktortypen dar. Seien also

$$\{\mathtt{Var}, \mathtt{Add}, \mathtt{Mul}, \mathtt{Num}\} \subset \mathcal{C}$$

Konstruktoren mit

$$\begin{aligned} \sigma_{\mathcal{C}}(\mathtt{Num}) &= 1 \\ \sigma_{\mathcal{C}}(\mathtt{Var}) &= 1 \\ \sigma_{\mathcal{C}}(\mathtt{Add}) &= 2 \\ \sigma_{\mathcal{C}}(\mathtt{Mul}) &= 2 \end{aligned}$$

Im Vergleich zu Haskell und MetaOCaml, wo jeder **V**-Term vom Typ `V` ist, sind verschiedene **V**-Terme in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ in der Regel von unterschiedlichem Typ. Zum Beispiel entspricht der AST zum Term $\mathtt{add}(x, \mathtt{mul}(y, y))$ dem Wert

$$\mathtt{Add}(\mathtt{Var}(\text{"}x\text{"}), \mathtt{Mul}(\mathtt{Var}(\text{"}x\text{"}), \mathtt{Var}(\text{"}y\text{"})))$$

vom Typ

$$\mathtt{Add}(\mathtt{Var}(\mathtt{String}), \mathtt{Mul}(\mathtt{Var}(\mathtt{String}), \mathtt{Var}(\mathtt{String})))$$

Die Struktur eines Terms ist sowohl im Typnamen, als auch im Wert sichtbar – genau so, wie wir das schon von den Polytyplisten her kennen.

Die Darstellungen sind jedoch nicht isomorph, da in der Typdarstellung Information über den Wert einer Konstanten verloren geht. Einem Termtyp sind in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ mehrere Terme zugeordnet; zum Beispiel werden sowohl $\mathtt{add}(3, 4)$, als auch $\mathtt{add}(12, 4)$ intern als Werte vom Typ $\mathtt{Add}(\mathtt{int}, \mathtt{int})$ dargestellt.

```
vInter := Λγ :: ∗.typecase γ ⟨Λδ :: ∗.δ → (String → int) → int⟩ of
                  default : Λα :: ∗.λx : α.λe : String → int.0
                      Num : Λα :: ∗.λf : α → (String → int) → int.
                            λn : Num(α).λe : String → int.
                            n.0
                      Var : Λα :: ∗.λf : α → (String → int) → int.
                            λv : Var(String).λe : String → int.
                            e(v.0)
                      Add : Λα :: ∗.λf0 : α → (String → int) → int.
                            Λβ :: ∗.λf1 : β → (String → int) → int
                            λa : Add(α, β).λe : String → int.
                            (f0 a.0 e) + (f1 a.1 e)
                      Mul : Λα :: ∗.λf0 : α → (String → int) → int.
                            Λβ :: ∗.λf1 : β → (String → int) → int
                            λa : Mul(α, β).λe : String → int.
                            (f0 a.0 e) ∗ (f1 a.1 e)
```

Abbildung 4.3: Ein Interpreter für **V** in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

Zur Implementierung des **V**-Interpreters greifen wir auf die strukturelle Typanalyse, als Gegenstück zur *case*-Analyse in Haskell und MetaOCaml, zurück (siehe Abbildung 4.3).

Die Funktion `vInter` generiert eine Funktion zur Auswertung eines **V**-Terms, der in interner Darstellung vorliegt.

Wie gehabt, werden **V**-Konstanten ausgewertet, indem man ihren Wert aus dem zugehörigen Konstruktor `Num` extrahiert und der Wert von Variablen ergibt sich durch Nachschlagen in der Umgebung. Etwas komplexer ist das Generieren einer Funktion zur Auswertung von `Add`- und `Mul`-Knoten. Zunächst werden Funktionen zur Auswertung der Argumente generiert. Das geschieht gewissermaßen automatisch durch den `typecase`-Mechanismus: Ausgehend von der Struktur der Argumenttypnamen und der `typecase`-Anweisung werden entsprechende Funktionen generiert und den Funktionen in den Behandlungsfällen zu `Mul` und `Add` in den Argumenten f_0 und f_1 übergeben. Eine Funktion zur Auswertung von `Mul`-Knoten läßt sich dann leicht konstruieren: Die Funktionen zur Auswertung der Operanden werden mit den Konstruktorkomponenten des `Mul`-Knotens als Eingabe versorgt, um anschließend die Ergebnisse dieser Berechnungen miteinander zu multiplizieren.

Trifft der Interpreter auf einen Wert, der nicht zur Sprache **V** gehört (`default`-*case*), so wird eine Funktion zurückgegeben, die auf jede Eingabe mit `0` reagiert – wie wir in Kürze sehen werden, tritt dieser Fehler niemals auf, da er mit Hilfe des Typsystems ausgeschlossen werden kann.

Die Funktion `runVProg` haben wir aus gutem Grund in Abbildung 4.3 ausgespart. In einem ersten Versuch könnte man `runVProg` vielleicht so realisieren:

```
runVProg := λx : int.λy : int.vInter ⟦(parse "add(x, mul(x, y))")⟧
                                     (mkEnv "y" y (mkEnv "x" x emptyEnv))
```

Wie in Haskell und MetaOCaml, übernimmt die Funktion `parse` einen **V**-Term als Zeichenkette, um daraus einen AST zu erzeugen.

Leider kann man `parse` so aber nicht in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ implementieren, da der Rückgabetyp abhängig vom Funktionsargument ist. In Haskell und MetaOCaml gibt es dieses Problem nicht, da jeder AST den gleichen Typ `V` hat. Um `parse` implementieren zu können, müssten wir System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ um Typen erweitern, die von Termen abhängen (man spricht von *dependent types*). Diese lassen sich aber nur realisieren, wenn man Typinformation zur Laufzeit verfügbar macht und neue Typen zur Laufzeit erzeugen kann. Das steht aber im krassen Widerspruch zu unserem Anspruch, Programme durch Typelimination zu übersetzen.

Die Implementierung eines **V**-Parsers in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ scheint damit unmöglich. Macht man `add`, `mul`, `x` und `y` jedoch zu Objekten (Funktionen) von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ – und läßt damit **V** zu einer Teilmenge von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ werden – kann man durch geschicktes Ausnutzen von `typecase` und `Typecase` sehr wohl einen **V**-Parser programmieren (siehe Abbildung 4.4).

Die Implementierung des Parsers ist auf mehrere Funktionen verteilt, und wie wir gleich sehen werden verdienen sie eigentlich nicht den Namen „Parser", da keine syntaktische, sondern vielmehr eine semantische Analyse vollzogen wird.

Bevor wir die einzelnen Funktionen im Detail besprechen, wollen wir einige Überlegungen zu den Auswirkungen und Anforderungen dieser Integrationstechnik anstellen.

Betrachten wir den Term

$$\texttt{add}\,[\![x, \texttt{mul}\,[\![x, y]\!]]\!]$$

Mit den Funktionen aus Abbildung 4.4 wird dieser zu einem gültigen System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Term, der – abgesehen vom Klammerpaar $[\![$ und $]\!]$ – exakt einem **V**-Term in konkreter Syntax gleicht.

Die Funktionen `add`, `mul`, `x` und `y` sind nun so konstruiert, daß aus diesem Term ein **V**-Term in abstrakter Notation (ein AST) entsteht; mit anderen Worten führt seine Auswertung zu einem Wert vom Typ

$$\texttt{Add}(\texttt{Var}(\texttt{String}), \texttt{Mul}(\texttt{Var}(\texttt{String}), \texttt{Var}(\texttt{String})))$$

Aus der Sicht des Programmierers wird die syntaktische Analyse (das *Parsing*) zu einem rein impliziten Vorgang und der **V**-Term kann ohne weitere Verarbeitung direkt an den Interpreter weitergereicht werden:

$$\begin{array}{ll}\texttt{runProg} := \lambda x : \texttt{int}.\lambda y : \texttt{int}.\texttt{vInter} & [\![\texttt{add}\,[\![x, \texttt{mul}\,[\![x, y]\!]]\!]]\!] \\ & (\texttt{mkEnv "y" y (mkEnv "x" x emptyEnv}))\end{array}$$

Wie man sieht, muß weder ein Parser aufgerufen werden, noch ist es notwendig, den **V**-Term explizit zu quotieren. **V**-Terme heben sich allein durch ihren Typ vom Rest eines System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Terms ab. Entspricht der Typ eines System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Terms der abstrakten Syntax eines **V**-Terms, so identifizieren wir ihn als **V**-Term in konkreter Notation.

Da wir die syntaktischen Objekte von **V** unmittelbar auf System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Funktionen abbilden, wird das eigentliche *Parsing* vom System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Übersetzer übernommen. Als Ergebnis liefern diese Funktionen `x`, `y`, `mul` und `add` AST-Knoten. Die **V**-Variablen `x` und `y` werden im AST als `Var`-Knoten dargestellt, so daß die System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Funktionen eben diese als Ergebnis zurückliefern.

Die Funktionen `mul` und `add` sind als polymorphe Funktionen implementiert, deren Definitionsbereich durch `JustPair` auf Paartypen eingeschränkt ist. Die Beschränkung auf Paartypen

$$
\begin{array}{rcl}
\texttt{Res}_{\mathbb{T}} & := & \Lambda\delta :: *.\texttt{Typecase}\ \delta\ \texttt{of} \\
& & \quad \texttt{default}: \ \Lambda\alpha :: *.\texttt{FALSE} \\
& & \quad \rightarrow: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K.\beta \\
\texttt{Arg}_{\mathbb{T}} & := & \Lambda\delta :: *.\texttt{Typecase}\ \delta\ \texttt{of} \\
& & \quad \texttt{default}: \ \Lambda\alpha :: *.\texttt{FALSE} \\
& & \quad \rightarrow: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K.\alpha \\
\texttt{marshall}_{\mathbb{T}} & := & \Lambda\delta :: *.\texttt{Typecase}\ \delta\ \texttt{of} \\
& & \quad \texttt{default}: \ \texttt{void} \rightarrow \texttt{FALSE} \\
& & \quad \texttt{int}: \ \texttt{int} \rightarrow \texttt{Num(int)} \\
\texttt{marshall}_{\mathbb{V}} & := & \Lambda\delta :: *.\texttt{typecase}\ \delta\ \ \langle\Lambda\gamma :: *.\texttt{marshall}_{\mathbb{T}}[\gamma]\rangle\ \texttt{of} \\
& & \quad \texttt{default}: \ \Lambda\alpha :: *.\lambda x : \texttt{void}.\texttt{FALSE} \\
& & \quad \texttt{int}: \ \lambda x : \texttt{int}.\texttt{Num}(x) \\
\texttt{check}_{\mathbb{T}} & := & \Lambda\delta :: *.\texttt{Typecase}\ \delta\ \texttt{of} \\
& & \quad \texttt{default}: \ \Lambda\alpha :: *.\texttt{marshall}_{\mathbb{T}}[\alpha] \\
& & \quad \texttt{Var}: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\texttt{Var(String)} \rightarrow \texttt{Var(String)} \\
& & \quad \texttt{Num}: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\texttt{Num(int)} \rightarrow \texttt{Num(int)} \\
& & \quad \texttt{Add}: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K. \\
& & \qquad \texttt{Add}(\texttt{Arg}_{\mathbb{T}}[\alpha'], \texttt{Arg}_{\mathbb{T}}[\beta']) \rightarrow \texttt{Add}(\texttt{Res}[\alpha'], \texttt{Res}[\beta']) \\
& & \quad \texttt{Mul}: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K. \\
& & \qquad \texttt{Mul}(\texttt{Arg}_{\mathbb{T}}[\alpha'], \texttt{Arg}_{\mathbb{T}}[\beta']) \rightarrow \texttt{Mul}(\texttt{Res}_{\mathbb{T}}[\alpha'], \texttt{Res}_{\mathbb{T}}[\beta']) \\
\texttt{check}_{\mathbb{V}} & := & \Lambda\delta :: *.\texttt{typecase}\ \delta\ \ \langle\Lambda\gamma :: *.\texttt{check}_{\mathbb{T}}[\gamma]\rangle\ \texttt{of} \\
& & \quad \texttt{default}: \ \Lambda\alpha :: *.\texttt{marshall}_{\mathbb{V}}[\alpha] \\
& & \quad \texttt{Var}: \ \Lambda\alpha :: *.\lambda f : \texttt{check}_{\mathbb{T}}[\alpha].\lambda v : \texttt{Var}(\alpha).v \\
& & \quad \texttt{Num}: \ \Lambda\alpha :: *.\Lambda f_0 : \texttt{check}_{\mathbb{T}}[\alpha]. \\
& & \qquad \lambda n : \texttt{Num}(\alpha).n \\
& & \quad \texttt{Add}: \ \Lambda\alpha :: *.\lambda f_0 : \texttt{check}_{\mathbb{T}}[\alpha]. \\
& & \qquad \Lambda\beta :: *.\lambda f_1 : \texttt{check}_{\mathbb{T}}[\beta]. \\
& & \qquad \lambda a : \texttt{Add}(\alpha, \beta).\texttt{Add}(f_0(a.0), f_1(a.1)) \\
& & \quad \texttt{Mul}: \ \Lambda\alpha :: *.\lambda f_0 : \texttt{check}_{\mathbb{T}}[\alpha]. \\
& & \qquad \Lambda\beta :: *.\lambda f_1 : \texttt{check}_{\mathbb{T}}[\beta]. \\
& & \qquad \lambda m : \texttt{Mul}(\alpha, \beta).\texttt{Mul}(f_0(m.0), f_1(m.1)) \\
\texttt{x} & := & \texttt{Var}(\text{"}x\text{"}) \\
\texttt{y} & := & \texttt{Var}(\text{"}y\text{"}) \\
\texttt{JustPair}_{\mathbb{T}} & := & \Lambda\delta :: *.\texttt{Typecase}\ \delta\ \texttt{of} \\
& & \quad \texttt{default}: \ \Lambda\alpha :: *.\texttt{void} \\
& & \quad \times: \ \Lambda\alpha :: *.\Lambda\alpha' :: *.\Lambda\beta :: *.\Lambda\beta' :: *. \times (\alpha, \beta) \\
\texttt{add} & := & \Lambda\alpha :: *.\lambda x : \texttt{JustPair}[\alpha].\texttt{check}_{\mathbb{V}}\ [\![\texttt{Add}(x.0, x.1)]\!] \\
\texttt{mul} & := & \Lambda\alpha :: *.\lambda x : \texttt{JustPair}[\alpha].\texttt{check}_{\mathbb{V}}\ [\![\texttt{Mul}(x.0, x.1)]\!]
\end{array}
$$

Abbildung 4.4: Parser für $\mathbf{V}$ in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

reicht jedoch nicht aus. Da `mul` und `add` Funktionen der Sprache **V** sind, müssen die Paarkomponenten auf gültige Fragmente von **V** eingeschränkt werden. Zwei Fälle sind zu unterscheiden:

- Die Komponenten sind keine gültigen **V**-Terme. Das kann z.B. der Fall sein, wenn `add` mit dem Argument $(\texttt{true}, \texttt{false})$ aufgerufen wird – die Sprache **V** kennt ja keine booleschen Konstanten
- Der Paartyp enthält Komponenten, die in gültige AST-Knoten gewandelt werden können. In der konkreten Syntax von **V** werden numerische Konstanten genauso dargestellt wie in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$. Im AST erscheinen sie jedoch als `Num`-Knoten. Wird `add` auf das Paar $(1, 2)$ angewendet, so müssen die Konstanten 1 und 2 in `Num`-Knoten eingebettet werden. Diesen Transport von Werten aus einer Sprache in eine andere nennt man auch **Marshalling**.

Das Marshalling wird von den Funktionen $\texttt{marshall}_{\mathbb{V}}$ und $\texttt{marshall}_{\mathbb{T}}$ realisiert. Abhängig vom System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Typ wird von $\texttt{marshall}_{\mathbb{V}}$ eine Funktion generiert, die Werte dieses Typs in gültige AST-Knoten konvertiert.

Für unseren Interpreter beschränkt sich der Datenaustausch auf `int`-Werte, die im abstrakten Syntaxbaum vom Konstruktor **Num** umschlossen werden. Für alle anderen System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ Typen wird eine Funktion generiert, die Argumente vom Typ `void` erwartet und somit praktisch niemals aufgerufen werden kann. Der Versuch einen Wert vom Typ `bool` in einem **V**-Term zu verwenden führt folglich zu einem Typfehler.

In die umgekehrte Richtung, also von **V** nach System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$, wird das Marshalling vom Interpreter bestimmt, indem er aus einem **V**-Term einen System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Wert (einen Wert vom Typ `int`) berechnet.

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Terme können Unterterme von **V**-Termen sein; d.h. es ist möglich beide Sprachen bis zu einem gewissen Grad zu mischen. Das Marshalling von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ nach **V** ist allein abhängig vom Typ eines System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Terms und folglich kann überall dort, wo in einem **V**-Term ein `int`-Wert stehen darf, ein beliebiger System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Term vom Typ `int` eingesetzt werden.

Abbildung 4.5 zeigt gültige und ungültige Kombinationen von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$- und **V**-Termen.

Konkrete Syntax (System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$)	Typ	Abstrakter **V**-Term
$\lambda x : \texttt{int}.x + 1$	$\texttt{int} \to \texttt{int}$	Nein
$\texttt{add}\,[\![(1, 2)]\!]$	$\texttt{Add}(\texttt{Num}(\texttt{int}), \texttt{Num}(\texttt{int}))$	Ja
$\texttt{vInter add}\,[\![(1, 2)]\!]$ e	`int` (vorausgesetzt $\vdash e : \texttt{String} \to \texttt{int}$)	Nein
$\texttt{add}\,[\![(3 * 4, 4)]\!]$	$\texttt{Add}(\texttt{Num}(\texttt{int}), \texttt{Num}(\texttt{int}))$	Ja
$\texttt{add}\,[\![((\lambda x : \texttt{int}.x + 2)3, 4)]\!]$	$\texttt{Add}(\texttt{Num}(\texttt{int}), \texttt{Num}(\texttt{int}))$	Ja
$1 + \texttt{add}\,[\![(1, 3)]\!]$	Typfehler!	—

Abbildung 4.5: Durch Mischen von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ und **V** können gültige und ungültige Funktionen entstehen.

Wie gesagt, reicht die Einschränkung der Funktionen `add` und `mul` auf Paartypen nicht aus. Es muß gewährleistet sein, daß es sich bei den konkreten Argumenten entweder um gültige **V**-Terme handelt, oder sie sich durch Marshalling in gültige **V**-Terme konvertieren lassen.

Das Funktionspaar $\texttt{check}_{\mathbb{V}}$ und $\texttt{check}_{\mathbb{T}}$ übernimmt genau diese Aufgaben. Die Idee ist, aus den Komponenten des an `mul` und `add` übergebenen Paars zunächst einen AST-Knoten zu generieren – ohne Rücksicht auf mögliche Fehler, weshalb man hier besser von einem „Kandidaten" spricht. Durch strukturelle Typanalyse wird aus dem Typ des Kandidaten von der Funktion $\texttt{check}_{\mathbb{V}}$ eine Funktion erzeugt, die den Kandidaten in einen gültigen AST-Knoten überführt. Ungültige Kandidaten werden durch geeignete Begrenzung des Definitionsbereiches von $\texttt{check}_{\mathbb{V}}$ herausgefiltert.

Die Einschränkung übernimmt das Funktionspaar $\texttt{check}_{\mathbb{T}}$ / $\texttt{marshall}_{\mathbb{T}}$: Knoten, die *nicht* mit den Konstruktoren `Var`, `Num`, `Add` oder `Mul` erzeugt wurden, fallen unter den `default`-*case*. Dann entspricht der Typ der Prüffunktion exakt dem Typ der Marshalling Funktion und ist somit nur auf `int`-Werte anwendbar.

Für `Var`- und `Num`-Knoten wird der Typ der Prüffunktion so gewählt, daß nur Knoten mit passendem Komponententyp akzeptiert werden (also `string` im Falle von `Var` und `int` im Falle von `Num`).

Für Additions- und Multiplikationsknoten ergibt sich der Typ der Prüffunktion induktiv aus den für die Konstruktorkomponenten generierten Prüffunktionstypen. Gesteuert vom `Typecase`-Mechanismus werden diese berechnet und den Behandlungsfällen für `Mul` und `Add` in den Argumenten α' und β' übergeben. Unter Anwendung der Hilfsfunktionen $\texttt{Arg}_{\mathbb{T}}$ und $\texttt{Res}_{\mathbb{T}}$ können Argument- und Ergebnistyp dieser Prüffunktion ermittelt werden. Der Argumenttyp der Prüffunktion für `Add` und `Mul` wird so konstruiert, daß nur solche Knoten akzeptiert werden, deren Komponenten mit den Prüffunktionen α' und β' kompatibel sind. Ebenso ergibt sich der Ergebnistyp aus dem Ergebnistyp der Prüffunktionen für die Komponenten.

Enthält z.B. ein `Add`-Knoten an einer bestimmten Position einen ungültigen Wert (z.B. die Konstante `false`), so erwartet die Prüffunktion als Argument einen Knoten, der genau an dieser Stelle einen Wert vom Typ `void` speichert. Zum Beispiel ergibt

$$\texttt{check}_{\mathbb{T}}[\texttt{Add}(\texttt{bool}, \texttt{int})]$$

eine Prüffunktion vom Typ

$$\texttt{Add}(\texttt{void}, \texttt{int}) \rightarrow \texttt{Add}(\texttt{FALSE}, \texttt{Num}(\texttt{int}))$$

Da es in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ keine Werte vom Typ `void` gibt, führt ein Aufruf dieser Funktion zwangsläufig zu einem Typfehler.

Die eigentliche Prüffunktion wird von der Funktion $\texttt{check}_{\mathbb{V}}$ generiert. Die Wirkung der einzelnen Behandlungsfälle sollte nach der Diskussion von $\texttt{check}_{\mathbb{T}}$ klar sein: Knoten, die eines Marshalling bedürfen, werden durch Marshalling konvertiert; `Num` und `Var` Knoten bleiben unverändert und im Falle von binären Knoten werden deren Komponenten analysiert.

Interessant ist hier die besondere Rolle der Funktion zur Berechnung des Ergebnistyps. Diese sorgt nämlich dafür, daß Prüffunktionen von vorn herein nur für korrekte Knoten generiert werden können. Soll z.B. ein Funktion zur Prüfung des Knotentyps $\texttt{Add}(\texttt{bool}, \texttt{int})$ erzeugt werden, stimmt der von $\texttt{parse}_{\mathbb{T}}$ berechnete Argumenttyp nicht mit dem Argumenttyp der Funktion im Behandlungsfall für `Add`-Knoten überein und es kommt zu einem Typfehler (siehe Typregel **T-typecase** auf Seite 65).

Als Konsequenz können nur gültige $\mathbf{V}$-Terme entstehen. Die Funktionen $\texttt{check}_{\mathbb{V}}$ und $\texttt{check}_{\mathbb{T}}$ übernehmen die Rolle des Typecheckers für $\mathbf{V}$ und garantieren, daß ausschließlich gültige

V-Terme entstehen können. Der Interpreter kann folglich nur auf korrekten **V**-Termen operieren und – wie bereits erwähnt – wird der `default`-*case* in der Funktion `vInter` niemals angesprungen.

In den Anhängen A.2 und A.3 zeigen wir leicht modifizierte Versionen von Parser und Interpreter und weisen nach, daß der partielle Auswerter den Term

$$\texttt{add}\,[\![(x, \texttt{mul}\,[\![(x, y)]\!])]\!]$$

tatsächlich zu einem AST

$$\texttt{Add}(\texttt{Var}(\text{"}x\text{"}), \texttt{Mul}(\texttt{Var}(\text{"}x\text{"}), \texttt{Var}(\text{"}y\text{"})))$$

reduziert und daß aus dem Interpreteraufruf

$$\texttt{vInter}\,[\![\;\; \texttt{add}\,[\![(x, \texttt{mul}\,[\![(x, y)]\!])]\!] \;\;]\!] \;\; e$$

das Residuum

$$\begin{array}{ll} (& \lambda m : \texttt{Add}(\texttt{Var}(\texttt{String})(\texttt{Var}(\texttt{String}), \texttt{Var}(\texttt{String})). \\ & \lambda e' : \texttt{String} \to \texttt{int}.e'(m.1) + (e'(m.0.0) * e'(m.1.1))) \\ ) & e \end{array}$$

hervorgeht.

Fassen wir noch einmal kurz zusammen: Zur Integration einer Sprache in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ bildet man die syntaktischen Konstrukte der einzubettenden Sprache auf System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Funktionen ab, die AST-Knoten als Argument übernehmen und im Ergebnis einen neuen AST generieren. Da sich die Struktur eines AST in seinem Typ abzeichnet, es also keinen einheitlichen Typ für ASTs gibt, müssen diese Funktionen polymorph sein. Damit werden sie jedoch zu allgemein. Um zu verhindern, daß Elemente, die nicht der einzubettenden Sprache angehören, erzeugt werden können, muß man die Argumenttypen durch Einsatz von `Typecase` einschränken – mit anderen Worten codiert man ein Typsystem für die eingebettete Sprache innerhalb des Typsystems der Hostsprache. Überladungsapplikation bildet schließlich den Schlüssel zu einer benutzerfreundlichen Darstellung der eingebetteten Sprache, da auf die für polymorphe Funktionen notwendige Typapplikation verzichtet werden kann.

Die lexikalische und die syntaktische Analyse werden vom System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Übersetzer „geerbt". Durch partielle Auswertung mit $\mathcal{T}$ ist garantiert, daß die semantische Analyse und die Codegeneration zur Übersetzungszeit stattfinden.

4.2.5 Vergleich der Ansätze

In den vorangegangenen Abschnitten haben wir an drei Beispielen gezeigt, wie man eine neue Programmiersprache in eine bestehende einbetten kann und wie man durch partielle Auswertung einen Übersetzer für diese Sprachen in Maschinensprache gewinnt. In diesem Abschnitt wollen wir die verschiedenen Ansätze anhand von ausgesuchten Kriterien miteinander vergleichen.

Syntax und Einbettung

In Haskell und MetaOCaml kann die Einbettung einer Sprache durch die Implementierung eines Interpreters realisiert werden. Programme der eingebetteten Sprache werden als Zeichenkette dargestellt, so daß man Sprachen mit nahezu beliebiger Syntax integrieren kann. Eingebettetes Programm und Hostprogramm sind klar voneinander abgegrenzt.

Sprachintegration mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ setzt voraus, daß die einzubettende Sprache eine Teilmenge von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ist. Die Zahl der integrierbaren Sprachen ist dadurch zwar geringer, dafür kann man aber auf das explizite Quotieren der eingebetteten Sprache verzichten, woraus der Vorteil erwächst, daß bestehende Werkzeuge (z.B. Syntax-Highlighter, Code-Folder, Debugger) grundsätzlich auch mit der eingebetteten Sprache zurechtkommen.

Die Menge der integrierbaren Sprachen ließe sich durch entsprechende Erweiterungen an System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ natürlich ausweiten. Zum Beispiel könnte man überladbare und benutzerdefinierte infix-Operatoren in die Sprachdefinition mit aufnehmen; ebenso denkbar wären komplexere Konstrukte wie z.B. eine `if`-Anweisung, die intern auf eine dreistellige Funktion zurückgeführt wird, oder aber auch die Abbildung der Funktionsapplikationen auf eine interne Funktion (siehe z.B. `operator()` in C++).

In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ können Hostsprache und eingebettete Sprache gemischt werden. Sofern ein System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Term vom Typ `int` ist bzw. durch die Funktion `marshall` geklärt ist, wie System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Typen im AST gespeichert werden, kann er Teilterm eines **V**-Terms sein. Die Umkehrung gilt jedoch nicht – zumindest nicht so direkt. Das für System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ sichtbare Ergebnis eines **V**-Terms ist ein AST und kein Wert, der aus der Interpretation dieses Terms entspringt. Bevor auf den Wert eines **V**-Terms zugegriffen werden kann, muß er durch expliziten Aufruf des Interpreters ausgewertet werden.

Lexikalische und syntaktische Analyse

Der Ansatz in Haskell und MetaOCaml verlangt nach der Implementierung eines Lexers (einem Unterprogramm zur Durchführung der lexikalischen Analyse) und nach einem Parser, der aus einem Programm in konkreter Syntax einen AST generiert.

Lexikalische und syntaktische Analyse werden bei der Einbettung mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ vom System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Compiler übernommen. Der Aufwand zur Erstellung des Parsers ist marginal und erschöpft sich in der Bereitstellung von geeigneten System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Funktionen. Da es in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ möglich ist, eingebettete und Hostsprache zu mischen, steckt der Großteil der Komplexität in der Beschreibung von gültigen ASTs, also der Implementierung eines Typsystems für die eingebettete Sprache.

Implementierung des Interpreters

In allen Ansätzen wird die Strukturanalyse eines Terms über *case*-Analyse und *pattern matching* realisiert. Die Ergebnisse des Interpreters sind jedoch verschieden: In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ wird eine System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Interpreterfunktion generiert; in MetaOCaml ist es Objectcode und in Haskell das Resultat des Interpretationsprozesses – also ein `Integer`-Wert.

Semantische Analyse

Wie wir gesehen haben, kann die Typüberprüfung in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ zur Übersetzungszeit erfolgen, da alle relevanten Daten im Typ des AST zur Verfügung stehen. Typüberprüfung von Host- und Gastsprache laufen zur selben Zeit ab.

In MetaOCaml kann man einen Typchecker ähnlich wie den *staged interpreter* realisieren, so daß die *case analysis* in Phase-N und die eigentliche Typüberprüfung zu einer späteren Phase ausgeführt wird. Typüberprüfung von Host- und Gastsprache finden aber zu unterschiedlichen Zeitpunkten statt.

In unserer Haskell-Variante wird der AST zur Laufzeit konstruiert, so daß eine Typüberprüfung der eingebetteten Sprache zur Übersetzungszeit der Hostsprache nicht möglich ist.

Typfehler in der Gastsprache werden in Haskell und MetaOCaml also viel später erkannt, als in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$, es sei denn, Operatoren der Gastsprache werden auf Operatoren der Hostsprache abgebildet, und es kommt dabei zu einem Typfehler.

Codegenerierung und partielle Auswertung

In Haskell kann der Programmierer keinen Einfluß auf den Vorgang der partiellen Auswertung nehmen und es gibt folglich keine Garantie, daß entsprechende Optimierungen durchgeführt werden. Der Interpreter wird vom Haskell-Compiler direkt in Maschinensprache übersetzt, so daß semantische- und *case*-Analyse zur Laufzeit erfolgen.

Anders in MetaOCaml und System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$: Beide Sprachen definieren einen mehrstufigen Auswertungsmechanismus, und man kann den Interpreter so implementieren, daß Teilaufgaben explizit von verschiedenen Auswertungsstufen abgearbeitet werden. Die partielle Auswertung des Interpreters unterliegt damit der Kontrolle des Programmierers und man kann sicher sein, daß bestimmte Optimierungen tatsächlich durchgeführt werden.

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ und MetaOCaml unterscheiden sich jedoch in der Zahl möglicher Ausführungsstufen und dem Zeitpunkt, zu dem eine Stufe ausgeführt wird. System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ist zweistufig, wobei sich die Abgrenzung der einzelnen Stufen aus der Trennung von Typsystem und Semantik ergibt, so daß ein explizites Quotieren zum Markieren von dynamischen oder statischen Werten entfallen kann. In der ersten Ausführungsphase werden Typausdrücke zu Normalformen reduziert und Überladungsapplikationen aufgelöst. Die Ausführung wird vom partiellen Auswerter $\mathcal{T}$ vollzogen, der sich am besten in Verzahnung mit dem Typechecker realisieren läßt. Die zweite Phase entspricht der Reduktion gemäß den Vorschriften der Relation $\rightarrow$.

Die strukturelle Analyse eines $\mathbf{V}$-Terms entspricht in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ der strukturellen Analyse des zugehörigen Typnamens. Partielles Auswerten des Analyseschritts kann allein auf der Ebene von Typberechnungen erfolgen. Das hat zwei vorteilhafte Konsequenzen: Zum einen sind statische und dynamische Variablen klar voneinander abgegrenzt – eine *binding time* Analyse kann entfallen, zum anderen ist garantiert, daß der partielle Auswerter terminiert, da wir System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ so konstruiert haben, daß die Typisierbarkeit eines Terms entscheidbar ist.

In MetaOCaml werden keine Berechnungen zur Compilezeit durchgeführt. Die Programmausführung kann vom Programmierer durch geeignete Annotationen explizit in mehrere Phasen aufgeteilt werden. Das Quotieren kommt einer manuellen *binding time*-Analyse gleich: Quotierter Code ist dynamisch, unquotierter Code ist statisch und durch Einsatz von *escape* können Teile von dynamischem Code als statisch markiert werden. Im Unterschied zu traditioneller

partieller Auswertung (und Multi-Level-Sprachen), ist die Ausführungszeit von dynamischem Code weder vorgegeben, noch wird dessen Ausführung implizit angestoßen.

Im Vergleich zu Haskell garantiert der Ansatz mit MetaOCaml und System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ daß das Parsing und die *case*-Analyse *vor* der eigentlichen Auswertung des zu interpretierenden Codes erfolgen. Allerdings hat MetaOCaml den Nachteil, daß das Ergebnis der partiellen Auswertung nicht konserviert wird: In Haskell und System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ manifestiert es sich im erzeugten Maschinencode, in MetaOCaml ist es dagegen flüchtig und wird am Ende eines Programmlaufs verworfen.

In MetaOCaml werden **V**-Terme in MetaOCaml-Objectcode übersetzt, der zu einem späteren Zeitpunkt durch Anwenden des *run*-Operators gestartet werden kann. Der Objectcode kann entweder von einer virtuellen Maschine interpretiert, oder aber durch einen *just-in-time* Compiler in Maschinensprache übersetzt und anschließend zur Ausführung gebracht werden. Im ersten Fall ergibt sich der Nachteil, daß die Ausführung von **V**-Termen wesentlich von den Eigenschaften des Objectcode-Interpreters abhängt. Im zweiten Fall hat man lediglich mit einem Effizienzverlust beim ersten Aufruf des Interpreters zu rechnen.

In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ werden **V**-Terme zunächst in die Hostsprache System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ übersetzt, um anschließend von einem Compiler in Maschinensprache überführt zu werden. Die eingebettete Sprache kann so von den spezifischen Optimierungen des Übersetzers profitieren.

Abbildung 4.6 faßt unsere Überlegungen zusammen.

Einen weiteren Vorteil von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ stellen wir im nächsten Abschnitt vor: Dadurch, daß man Subsprachen durch Typannotationen aus einem System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Programm extrahieren kann, ist es möglich, Optimierungen für Terme über ganz bestimmten Datentypen zu System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ hinzuzufügen, ohne den Übersetzer erweitern zu müssen. System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ eignet sich damit zur Realisierung von aktiven Bibliotheken [172].

4.3 Exkurs: Benutzerdefinierte Optimierungen mit System $\mathbf{F}^{\mathbf{SA}}_{\omega,1}$

Nehmen wir an, wir könnten binäre Operatoren in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ selbst definieren und überladen und müßten ein Programm erstellen, das zahlreiche Berechnungen über dreistelligen Vektoren erfordert.

Vektoren lassen sich durch Konstruktortypen darstellen. Da wir am Namen des Konstruktors nicht interessiert sind, greifen wir auf den speziellen Konstruktor $\times$ zurück, für den wir vereinbart hatten, daß er in der konkreten Syntax ausgelassen werden kann. Wenn wir nun zwei Infix-Operatoren `.+` und `.*` zur komponentenweisen Addition und Multiplikation bereitstellen, lassen sich Vektoroperationen sehr anschaulich und lesbar formulieren:

$$(a_1, b_1, c_1).*(a_2, b_2, c_2).+(a_3, b_3, c_3)$$

Gehen wir davon aus, Vektoroperationen seien komponentenweise definiert. Dann würde dieser Term reduziert, indem zunächst die ersten beiden Vektoren multipliziert werden. Im Ergebnis entsteht der Vektor

$$V_{temp} := (a_1 * a_2, b_1 * b_2, c_1 * c_2)$$

	Haskell	MetaOCaml	System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$
Darstellung AST	Summentyp	Summentyp	Mehrere Produkttypen
Analyse des AST	`case` (Laufzeit)	`case` (Laufzeit / Stage 0)	`Typecase` (Übersetzungszeit)
Einschränkung einer Funktion auf AST-Typ	Summentyp (statische Semantik)	Summentyp (statische Semantik)	`Typecase` (Programmierer / statische Semantik)
Lexikalische Analyse	Programmierer (Laufzeit)	Programmierer (Laufzeit / Stage 0)	System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ Grammatik (Übersetzungszeit)
Syntaktische Analyse	Parser	Parser	System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Parser / `Typecase`
Generierter Code	–	MetaOCaml Objectcode	System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ (nach Stufe 0) / Maschinencode durch Übersetzer
Ausführung von eingebettetem Code	Laufzeit	Laufzeit / Stage n	Laufzeit
Optimierungen	Explizite Manipulation des AST (Laufzeit)	MetaOCaml-Compiler Explizite Manipulation des AST (Laufzeit / Stage n-x)	System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Compiler Explizite Manipulation des AST (Übersetzungszeit / Laufzeit)

Abbildung 4.6: Vergleich der verschiedenen Interpreter.

Es ist klar, daß für dieses Zwischenergebnis Speicherplatz reserviert und initialisiert werden muß. Nachdem der Vektor V_{temp} erzeugt wurde, wird mit der Reduktion fortgefahren. Der Vektor (a_3, b_3, c_3) wird zum Vektor V_{temp} addiert, wobei der Vektor

$$V_R := (a_1 * a_2 + a_3, b_1 * b_2 + b_3, c_1 * c_2 + c_3)$$

entsteht. Der temporäre Vektor V_{temp} wird jetzt nicht mehr benötigt und könnte von einem *garbage collector* [89] freigegeben werden.

Erzeugen, Initialisieren und Freigeben von V_{temp} kostet Zeit und vorübergehend auch Speicherplatz. Offenbar wäre es effizienter, wenn man V_R direkt berechnet.

In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ können wir eine entsprechende Optimierung realisieren, ohne in den Übersetzer eingreifen zu müssen. Dazu fassen wir Ausdrücke über Vektortypen als domänenspezifische Sprache auf, und implementieren Parser und semantische Analyse analog zu **V**.

Den Interpreter zerlegen wir in eine interne Funktion `interVecCol` und eine Funktion `interVec`, die die Schnittstelle zum Benutzer darstellt. Die Funktion `interVecCol` übernimmt neben dem auszuwertenden Term als zusätzliches Argument einen Index, der bestimmt, welche Spalte des Ergebnisvektors berechnet werden soll.

Die Indizes kodieren wir jedoch nicht als `int`-Werte, sondern durch die nullstelligen Konstruktoren `EINS`, `ZWEI` und `DREI`. Durch diesen kleinen Trick kann eine Verzweigung in Abhängigkeit vom Index bereits zur Übersetzungszeit von `typecase` aufgelöst werden. Der Aufruf

$$\texttt{interVecCol ZWEI } [\![(a_1, b_1, c_1).*(a_2, b_2, c_2).+(a_3, b_3, c_3))]\!]$$

liefert dann das Ergebnis der zweiten Zeile der Vektoroperation, also den Wert $b_1 * b_2 + b_3$.

Die Funktion `interVec` übernimmt nun einen Vektorausdruck und berechnet diesen Spalte für Spalte durch Aufruf von `interVecCol` (die Funktion `JustVec` möge α auf gültige Vektortermtypen einschränken).

$$\begin{aligned}\texttt{interVec} = \Lambda\alpha :: *.\lambda v : \texttt{JustVec}[\alpha].(\ & \texttt{interVecCol EINS } [\![v]\!]\,,\\ & \texttt{interVecCol ZWEI } [\![v]\!]\,,\\ & \texttt{interVecCol DREI } [\![v]\!])\end{aligned}$$

Betrachten wir ein weiteres Beispiel aus dem Bereich der Vektorarithmetik. Gegeben sei der Ausdruck

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} * (y_1, \cdots, y_n) * \begin{pmatrix} z_1 \\ \vdots \\ z_n \end{pmatrix}$$

Die Assoziativität der Matrixmultiplikation läßt einem die Wahl, welche Multiplikation zuerst ausgeführt werden soll. Die meisten Übersetzer interpretieren die Multiplikation linksassoziativ, so daß mit der Multiplikation des Spaltenvektors x mit dem Zeilenvektor y begonnen wird. Dabei entsteht eine $n \times n$ Matrix, die von rechts mit dem Spaltenvektor z multipliziert wird, wobei im Ergebnis ein Spaltenvektor entsteht. Insgesamt sind $n + n^2$ Multiplikationen erforderlich.

Rechnet man zunächst die zweite Multiplikation aus, entsteht aus dem Produkt des Zeilenvektors y und des Spaltenvektors z ein Skalar, daß anschließend mit dem Spaltenvektor x multipliziert wird. In diesem Fall sind insgesamt nur $2 * n$ Multiplikationen notwendig – das Verfahren ist also erheblich effizienter.

Faßt man Operationen über Vektoren als *Domain Specific Language* (DSL) auf und macht den Unterschied zwischen Zeilen- und Spaltenvektor im Typ der Objekte sichtbar, können entsprechende Optimierungen in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ zur Übersetzungszeit vorgenommen werden.

Wie man sich vielleicht vorstellen kann, sind auch komplexere Optimierungen möglich, bei denen Terme in abstrakter Notation zur Übersetzungszeit umstrukturiert werden. Dazu können allerdings Operationen notwendig werden, oder Effekte auftreten, die der partielle Auswerter $\mathcal{T}$ nicht wegoptimieren kann (z.B. die Elimination von temporären Werten, das Extrahieren von gemeinsamen Teilausdrücken etc.). In diesen Fällen wird man abhängig von den Optimierungen, die der System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Übersetzer zur Verfügung stellt. Im zweiten Teil dieser Arbeit werden wir an einer praktischen Anwendung sehen, daß sich der Mehraufwand jedoch in Grenzen hält und in den meisten Fällen sogar völlig egalisiert werden kann.

4.4 Interpreter für Sprachen mit Rekursion - System $\mathbf{F}^{\mathbf{SA}}_{\omega,2}$

V ist eine sehr einfache Programmiersprache ohne Kontrollstrukturen und Schleifen. In diesem Abschnitt wollen wir der Frage nachgehen, ob sich auch komplexere Sprachen in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ integrieren lassen. Wir werden zeigen, daß es dazu einer kleinen Spracherweiterung bedarf, die uns zwingt, auf einige Eigenschaften von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ zu verzichten.

Zunächst erweitern wie **V** um benutzerdefinierte Funktionen, die auch rekursiv definiert sein dürfen, sowie um eine `If`-Anweisung. Die so erweiterte Sprache nennen wir $\mathbf{V}_{\mathrm{Rek}}$ (siehe Abbildung 4.7).

Ein $\mathbf{V}_{\mathrm{Rek}}$-Programm besteht aus mehreren Funktionsdefinitionen und einem auszuwertenden Term. Die Semantik eines $\mathbf{V}_{\mathrm{Rek}}$-Programmes entspricht, intuitiv gesprochen, dem Prinzip „Einsetzen und Ausrechnen".

Die syntaktische Integration von $\mathbf{V}_{\mathrm{Rek}}$ in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ wollen wir nicht weiter vertiefen. Wir könnten System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ natürlich leicht um zwei Funktionssymbole „`=`" und „`;`" erweitern, die in der konkreten Syntax infix geschrieben werden dürfen, um so die Argumentliste an `define` zu parsen. Ebenso einfach könnten wir ein infix-Symbol `in` hinzunehmen und eine Funktion `If` vereinbaren, so daß sich $\mathbf{V}_{\mathrm{Rek}}$-Programme als Teilmenge von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ darstellen lassen.

Wir wollen uns hier ausschließlich auf die Umsetzung der Semantikregel **F-Call** konzentrieren, denn alle anderen Regeln lassen sich nahezu analog zum **V**-Interpreter umsetzen.

Der Aufruf einer benutzerdefinierten Funktion f_i wird reduziert, indem zunächst Normalformen N_1 bis N_n zu den Argumenten V_{a1} bis V_{an} bestimmt werden. Anschließend wird die rechte Seite der Funktionsdefinition (der Term V_i) instanziiert; d.h. die Parameter x_{i1} bis x_{in} werden durch die zugehörigen Argumente N_1 bis N_n ersetzt, und die Normalform des neu entstandenen Terms bestimmt.

Die Reduktion der Funktionsargumente läßt sich genau wie bei der Addition und der Multiplikation durch Ausnutzen des `typecase`-Mechanismus bewerkstelligen. Das ist möglich, da das von `typecase` vorgegebene Rekursionsmuster exakt zum Rekursionsmuster des Interpreters paßt. Die Operation selbst kann aber nicht mehr unmittelbar ausgeführt werden: Die instan-

Syntax

$\mathbf{V}_{\mathrm{Rek}}$

$V ::=$		**V-Terme**
	x	Variable
	f(V,$\cdots$,V)	Funktionsaufruf
	N	Ganze Zahl
	$\texttt{add}(V, V)$	Addition
	$\texttt{mul}(V, V)$	Multiplikation
	$\texttt{If}(V, V, V)$	Einfache Selektion
$V_{Prog} ::=$	$\texttt{define}(f_1(x_1 1, \cdots, x_1 m) = V_1;$	Skript
	$\cdots;$	
	$f_n(x_n 1, \cdots, x_n m) = V_n$	
	$)\ \texttt{in}\ V$	

$V \Downarrow N$

$$N \Downarrow N \quad \textbf{(Num)}$$

$$\frac{V_1 \Downarrow N_1 \quad V_2 \Downarrow N_2}{\texttt{op}(V_1, V_2) \Downarrow \overline{N_1 op N_2}} \quad \textbf{(BinOp)}$$

$$\frac{V_C \Downarrow 0 \quad V_T \Downarrow N_T}{\texttt{If}(V_C, V_T, V_E) \Downarrow N_T} \quad \textbf{(If-True)}$$

$$\frac{V_1 \Downarrow N \quad N \neq 0 \quad V_E \Downarrow N_E}{\texttt{If}(V_C, V_T, V_E) \Downarrow N_E} \quad \textbf{(If-False)}$$

$$\frac{\begin{array}{c} V_{a1} \Downarrow N_1 \quad \cdots \quad V_{an} \Downarrow N_n \\ V_i[N_1/x_1, \cdots, N_n/x_n] \Downarrow N_r \end{array}}{f_i(V_{a1}, \cdots, V_{an}) \Downarrow N_r} \quad \textbf{(F-Call)}$$

Abbildung 4.7: Syntax und Semantik von $\mathbf{V}_{\mathrm{Rek}}$, einer einfachen Programmiersprache mit rekursiven Funktionen.

ziierte rechte Seite muß vom Interpreter selbst reduziert werden; d.h. er muß explizit rekursiv aufgerufen werden.

Das Rekursionsmuster des Interpreters ist nicht mehr allein von der Struktur eines Terms bestimmt und paßt damit nicht mehr mit dem Rekursionsmuster von `typecase` und `Typecase` zusammen!

Machen wir uns das an der prototypischen Implementierung eines $\mathbf{V}_{\mathrm{Rek}}$-Interpreters noch einmal klar – siehe Abbildung 4.8.

Auf die interne Darstellung eines $\mathbf{V}_{\mathrm{Rek}}$-Skripts wollen wir nicht näher eingehen. Den Typ des AST zu einem $\mathbf{V}_{\mathrm{Rek}}$-Skript bekommt die Funktion `vRekInter` im Typargument ξ übergeben, das Skript selbst im Wertargument p. Die eigentliche Interpreterfunktion heißt `calc` und ergibt sich durch Fixpunktberechnung. `calc` generiert ausgehend von einem Typ α eine Funktion, die einen Wert vom Typ α (einen $\mathbf{V}_{\mathrm{Rek}}$-Term in abstrakter Notation) bei gegebener Umgebung *Env* (Kurzschreibweise für $\texttt{String} \rightarrow \texttt{int}$) zu einem `int`-Wert reduziert.

Der Einfachheit halber gehen wir davon aus, daß jeder benutzerdefinierten Funktion ein eigener Konstruktor zugeordnet ist und geben exemplarisch nur die Reduktionsvorschrift für einen n-

```
vRekInter := Λξ :: ∗.λp : ξ.
             fix λ calc : ∀α.α → Env → int
             Λη :: ∗.
             typecase η ⟨Λδ :: ∗.δ → Env → int⟩ of
               Num :   Λα :: ∗.λf : α → Env → int.λn : Num(α).λe : Env.n.0
               Var :   Λα :: ∗.λf : α → Env → int.λv : Var(α).λe : Env.e(n.0)
               Add :   Λα :: ∗.λf1 : α → Env → int.
                       Λβ :: ∗.λf2 : β → Env → int
                       λa : Add(α, β).λe : Env.
                       (f1 a.0 e) + (f2 a.1 e)
               Mul :   Λα :: ∗.λf1 : α → Env → int.
                       Λβ :: ∗.λf2 : β → Env → int
                       λa : Mul(α, β).λe : Env.
                       (f1 a.0 e) ∗ (f2 a.1 e)
               UFunc : Λα1 :: ∗.λf1 : α1 → Env → int
                       ⋮
                       Λαn :: ∗.λfn : αn → Env → int
                       λs.UFunc(α1, · · · , αn).
                       λe : Env.
                       let s′ = UFunc(f0 s.0 e, · · · , fn s.n e)
                           in calc [ Inst_T[ξ][UFunc(int, · · · , int)] ]
                                   rhs_V[ξ](p)[UFunc(int, · · · , int)] s′
                                   extendEnv[ξ] p [UFunc(int, · · · , int)] s′ e
```

Abbildung 4.8: Interpreter für $\mathbf{V}_{\mathrm{Rek}}$ in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

stelligen Konstruktor `UFunc` an.

Wird `UFunc` aufgerufen, müssen zunächst die Funktionsargumente ausgewertet werden. Die hierzu notwendigen Funktionen stellt der induktive `typecase`-Mechanismus in den Argumenten f_0 bis f_n zur Verfügung. Diese werden zusammen mit der Umgebung e auf die Komponenten des `UFunc`-Knotens angewendet und in einem neuen `UFunc`-Knoten gespeichert, der den nachfolgenden Berechnungsschritten über einen `let`-Ausdruck zugänglich gemacht wird.

Jetzt beginnt der interessante Part der Auswertung, bei der ein (polymorph) rekursiver Aufruf des Interpreters notwendig wird. Um einen Interpreter für `UFunc` zu generieren, benötigt man den Typ der rechten Seite der Funktionsdefinition von `UFunc`, wobei die gebundenen Variablen durch die konkreten Argumenttypen ersetzt werden. Diese Aufgabe wird von der Funktion $\mathtt{Inst}_{\mathbb{T}}$ übernommen, die aus der Typdarstellung des Skripts, also dem Typ ξ, die rechte Seite von `UFunc` extrahiert und die gebundenen Variablen durch den Typ `int` ersetzt (in $\mathbf{V}_{\mathrm{Rek}}$ gibt es ja nur den Typ `int`).

Der dabei entstehende Termtyp wird der Funktion `calc` übergeben, die dann einen geeigneten Interpreter generiert. Dieser Interpreter erwartet zwei Argumente: Zum einen die rechte Seite der Definition von `UFunc` in Wertdarstellung, zum anderen eine Umgebung, die den gebundenen Variablen von `UFunc` die gerade berechneten Argumentwerte zuweist. Die Umgebung entsteht durch Erweiterung der Umgebung e, durch Anwenden der Funktion `extendEnv`.

`extendEnv` erhält in den ersten beiden Argumenten den Skripttyp und das Skript, um so Zugriff

auf die Definition von `UFunc` zu erhalten. Als drittes Argument wird ihr ein `UFunc`-Knotentyp übergeben, dessen Komponenten die konkreten Argumenttypen widerspiegeln und im vierten Argument s', einen Wert des Typs, dessen Komponenten gerade die ausgewerteten Argumente des Aufrufs von `UFunc` ausmachen. Im fünften und letzten Argument wird schließlich die zu erweiternde Umgebung e übergeben.

In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ können wir `vRekInter` nicht implementieren, da wir polymorphe Rekursion aus gutem Grund ausgeschlossen haben.

Um eine Übersetzung von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ durch Typelimination zu ermöglichen, müssen sämtliche Typberechnungen zur Übersetzungszeit erfolgen. Wird eine polymorphe Funktion mit Hilfe von `fix` definiert, so müssen wir sie vor der Übersetzung „monomorphisieren".

Das ist leider nicht ganz so einfach, wie es scheint. In einem ersten Ansatz könnten wir versuchen, partiell auszuwerten, indem wir `fix` anwenden, anschließend die Typapplikation auswerten und mit der Reduktion des entstehenden Terms fortfahren. Der Haken an der Sache ist, daß dieses Verfahren nicht terminiert, da jede Anwendung von `fix` mindestens einen neuen `fix`-Term generiert, den der partielle Auswerter als Redex ansieht[2].

Um die Terminierung von $\mathcal{T}$ garantieren zu können bleiben Fixpunktberechnungen von der partiellen Auswertung nahezu unberührt und es wird lediglich der an `fix` übergebene Term reduziert, da dieser Typapplikationen, `typecase`- oder `Typecase`-Terme enthalten könnte:

$$\mathcal{T}[\![\texttt{fix}\ t]\!] = \texttt{fix}\ [\![t]\!]$$

Wir müssen daher eine andere Strategie verfolgen. Zunächst dehnen wir die Verwendung von `fix` auf polymorphe Funktionen aus, die nur von einem Typ abhängen. Den entsprechend erweiterten Kalkül nennen wir System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$.

System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$

$$\frac{\Gamma \vdash f : \forall\alpha :: *.T \to \forall\alpha :: *.T \quad \Gamma \vdash T :: *}{\Gamma \vdash \texttt{fix}\ f : \forall\alpha :: *.T} \quad \textbf{(T-Ext-Fix)}$$

$$\texttt{fix}\ \lambda f : \forall\alpha :: *.T.t \to t[\texttt{fix}\ \lambda f : \forall\alpha :: *.T.t/f] \quad \textbf{(E-Ext-Fix)}$$

Die Beschränkung auf einen Typparameter ist nicht unbedingt erforderlich, erlaubt uns aber eine einfachere Darstellung.

Betrachten wir die Funktionsdefinition

$$\texttt{recFunc} := \texttt{fix}\ \lambda f : \forall\alpha.T.\Lambda\alpha :: *.t$$

sowie die Applikation $\texttt{recFunc}[T]$.

Nehmen wir einmal an, wir würden alle Typen kennen, die in einer Aufrufspur zu $\texttt{recFunc}[T]$ als Argumente an `recFunc` auftreten. Das heißt, wie kennen alle Elemente des Definitionsbereichs von `recFunc`, die für die Berechnung von $\texttt{recFunc}[T]$ relevant sind.

Sei

$$\Delta(\texttt{recFunc}[T]) = \{T_0, \cdots, T_n\}$$

[2] Es sei denn, die in Frage stehende Funktion ist nicht wirklich rekursiv.

(mit $\not\exists i, j \in \{0, \cdots, n\}, i \neq j : T_i \equiv T_j$) diese endliche Menge. Dann können wir die polymorphe Funktionsdefinition von `recFunc` in zwei Schritten in ein System von n monomorphen Funktionen umwandeln, die sich gegenseitig aufrufen (engl.: *mutual recursive functions*):

1. Bilde $F(t[T_0/\alpha], \cdots, t[T_n/\alpha])$ (als Konstruktorwert). F enthält alle relevanten Instanzen von t und ist vom Typ $F(T[T_0], \cdots, T[T_n])$.
2. Für alle $i \in \{0, \cdots, n\}$: ersetze die rekursiven Aufrufe so, daß sie sich auf die zugehörigen Komponenten in F beziehen; d.h alle Aufrufe $f[T]$ in $F.i$ (mit $T \equiv T_j$) werden durch $F.j$ ersetzt.

Nach dem zweiten Schritt enthält t garantiert keine Referenzen mehr auf `recFunc`. Diese Garantie ist möglich, da wir polymorphe Funktionen nicht als Funktionsargumente erlauben (prädikativer Kalkül!) und alle Typapplikationen, in die `recFunc` involviert war, durch korrespondierende Komponenten von F ersetzt haben.

Sämtliche Instanzen von `recFunc`, die bei der Berechnung von `recFunc`$[T]$ entstehen können, finden sich jetzt als Komponenten im Konstruktor F wieder. Die einzelnen Funktionen erhalten durch die Position im Konstruktor gewissermaßen einen eindeutigen Namen – der Konstruktor `F` simuliert in diesem Zusammenhang also einen Namensraum.

Offenbar ist die Funktion

$$\mathtt{f}_{\mathrm{Rek}} := \lambda f : F(T[T_0], \cdots, T[T_n]).F(t[T_0/\alpha], \cdots, t[T_n/\alpha])$$

vom Typ

$$F(T[T_0], \cdots, T[T_n]) \to F(T[T_0], \cdots, T[T_n])$$

und wir können `fix` auf sie anwenden, um das rekursive Gleichungssystem schrittweise auszurechnen.

Das Ergebnis der Anwendung von `recFunc` auf den Typ T gewinnt man durch Projektion auf die Komponente j, die initial durch Instanziieren mit T konstruiert wurde:

$$\mathtt{recFunc} \quad = \quad (\mathtt{fix}\ f_{Rek}).j$$

Es fehlt ein Verfahren zur Bestimmung der Aufrufspur von $f[T]$. Ein einfaches Verfahren, das allerdings nicht immer terminiert, startet mit einer Menge von Typen Δ, die anfänglich nur den Typ T enthält. Dann wird f mit T instanziiert und es werden parallel alle möglichen Rechenpfade verfolgt (man geht also z.B. in einem `if`-Term sowohl in den else-, als auch den then-Zweig) und es werden die Typen zu Δ hinzugefügt, die im Rahmen eines rekursiven Aufrufs an f übergeben werden. Das Prozedere wird für alle Typen, die neu zu Δ hinzugekommen sind wiederholt, bis Δ nicht weiter wächst.

Es lassen sich leicht Beispiele finden, in denen dieses Verfahren nicht terminiert. In einer solchen Situation würde der partielle Auswerter natürlich ebenfalls nicht terminieren und wir verlieren eine zentrale Eigenschaft von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$. Als einziger Ausweg bleibt, das Verfahren

so zu erweitern, daß unendliche Expansionen automatisch erkannt werden, um dann zur Übersetzung der in Frage stehenden Funktion auf eine Übersetzungsstrategie ohne Typelimination umzuschalten – als Konsequenz würde man natürlich an Laufzeiteffizienz einbüßen.

Typerhaltung unter Reduktion und Fortschritt der Reduktion bleiben von der Erweiterung jedoch unberührt, genauso wie die Entscheidbarkeit der Typisierbarkeit eines Terms.

In Haskell und MetaOCaml lassen sich Sprachen mit rekursiven Funktionen prinzipiell einfacher integrieren. Da alle AST-Knoten denselben Typ haben, ist keine polymorphe Rekursion erforderlich und es kann grundsätzlich durch Typelimination übersetzt werden. Probleme ergeben sich aber im Zusammenhang mit der partiellen Auswertung, weil man genau wie bei System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ darauf achten muß, daß der partielle Auswerter terminiert. In MetaOCaml liegen entsprechende Vorkehrungen allein in der Verantwortung des Programmierers. Im Zweifelsfall muß man auf die partielle Auswertung von rekursiven Aufrufen verzichten – in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ist sie jedoch ein Muß, wenn man durch Typelimination übersetzen will.

4.5 Interpreter für Sprachen mit komplexen Typsystemen - System $\mathbf{F}^{\mathbf{SA}}_{\omega,3}$

Programmiersprachen, die wir bisher in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ integriert haben, zeichneten sich alle durch ein sehr einfaches Typsystem aus.

Was ist aber, wenn man Sprachen integrieren will, die ein strengeres, oder sogar mächtigeres Typsystem aufweisen, als es von System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ vorgeben ist? Dann reichen die Möglichkeiten von System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ unter Umständen nicht aus, da die Typlevel-Sprache nicht berechnungsuniversell ist.

Zum Beispiel fehlt es an einer brauchbaren Möglichkeit, die Äquivalenz von Typen explizit testen zu können. Eine entsprechende Operation ließe sich zwar mit `Typecase` implementieren, wäre jedoch extrem aufwendig und nur schwer erweiterbar.

Ein weiteres Manko ist, daß Typfunktionen bis jetzt nicht rekursiv definiert sein dürfen. Die einzige Möglichkeit zur Rekursion besteht in der Ausnutzung von `Typecase`, aber am Beispiel von rekursiven Interpretern haben wir bereits gesehen, daß sich nicht alle Funktionen durch strukturelle Induktion über den Aufbau des AST ausdrücken lassen.

Wir erweitern daher System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ zu einer vollwertigen zweistufigen Sprache System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ (siehe Abbildung 4.9), die auf beiden Stufen berechnungsuniversell ist. Später, in der praktischen Anwendung von System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$, werden wir sehen, daß eine entsprechende Ausdrucksstärke erforderlich ist, um zum Beispiel Sprachen, die polymorphe Rekursion unterstützen, integrieren zu können.

Es ist klar, daß mit dem so erweiterten Kalkül eine weitere Eigenschaft von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ verloren geht. Durch die Einführung des Fixpunktkombinators für Typen kann nicht mehr garantiert werden, daß $\rightsquigarrow$ terminiert und folglich kann man nicht mehr in allen Fällen entscheiden, ob ein Term typsicher ist.

Fortschritt und Typerhaltung unter Reduktion werden aber auch von System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ erfüllt.

$\boxed{\Gamma \vdash T :: K}$ System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$

$$\frac{\Gamma \vdash T_C :: * \quad \Gamma \vdash T_T :: K \quad \Gamma \vdash T_E :: K}{\Gamma \vdash \texttt{If}(T_C, T_T, T_E) : K} \textbf{ (K-TIf)}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash \texttt{Eq}(T_1, T_2) :: *} \textbf{ (K-TEq)} \qquad \frac{\Gamma \vdash T :: * \Rightarrow K}{\Gamma \vdash \texttt{Fix}\ T :: K} \textbf{ (K-TFix)}$$

$\boxed{T_1 \rightsquigarrow T_2}$

$$\frac{T_C \rightsquigarrow \texttt{TRUE} \quad T_T \rightsquigarrow T}{\texttt{If}\ T_C\ \texttt{then}\ T_T\ \texttt{else}\ T_E \rightsquigarrow T} \textbf{ (QR-If-True)}$$

$$\frac{T_C \rightsquigarrow \texttt{FALSE} \quad T_E \rightsquigarrow E}{\texttt{If}\ T_C\ \texttt{then}\ T_T\ \texttt{else}\ T_E \rightsquigarrow E} \textbf{ (QR-If-False)}$$

$$\frac{T_1 \rightsquigarrow T \quad T_2 \rightsquigarrow T}{T_1 \equiv T_2 \rightsquigarrow \texttt{TRUE}} \textbf{ (QR-EquivTrue)}$$

$$\frac{T_1 \rightsquigarrow T_1' \quad T_2 \rightsquigarrow T_2' \quad T_1' \neq T_2'}{T_1 \equiv T_2 \rightsquigarrow \texttt{FALSE}} \textbf{ (QR-EquivFalse)}$$

$$\frac{T \rightsquigarrow T'}{\texttt{Fix}\ T \rightsquigarrow \texttt{Fix}\ T'} \textbf{ (QR-Fix)}$$

$$\texttt{Fix}\ \Lambda\alpha :: *.T \rightsquigarrow T[\texttt{Fix}\ \Lambda\alpha :: *.T/\alpha] \textbf{ (QR-FixBeta)}$$

Abbildung 4.9: Erweiterung der Typsprache von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ zu einer berechnungsuniversellen Sprache.

4.6 Die System F_ω^{SA}-Sprachhierarchie

Mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$, System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ und System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ haben wir eine Hierarchie von Sprachen vorgestellt, die bezüglich der Einbettung von Programmiersprachen unterschiedliche Eigenschaften aufweisen. In dieser Sektion wollen wir unsere Erkenntnisse kurz zusammenfassen.

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ eignet sich zur Einbettung von einfachen Sprachen ohne Rekursion und mit Typsystemen, die sich allein durch strukturelle Induktion über den Termaufbau realisieren lassen. Trotz dieser Einschränkungen gibt es insbesondere im Hinblick auf Programmbibliotheken zahlreiche Anwendungen für System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

Es gibt viele Beispiele von Programmbibliotheken, die man auch als DSL verstehen kann. Exemplarisch seien Matrix-Bibliotheken, Bibliotheken aus dem Bereich der Bildverarbeitung und algorithmische Skelette [34, 17, 94] genannt. Jede dieser Bibliotheken stellt einen Satz von Funktionen zur Verfügung, die auf den Objekten ihrer *domain* (also Matrizen und Vektoren, Bilddaten und Feldern) operieren und meist können diese Funktionen auch verschachtelt angewendet werden.

Der Übersetzer hat in der Regel kein Wissen über die Struktur dieser Daten und kennt die Semantik der Bibliotheksfunktionen nicht. Folglich kann er bestimmte domänenspezifische Optimierungen nicht durchführen und sie müssen manuell vom Programmierer berücksichtigt werden.

Wie wir in Abschnitt 4.3 skizziert haben, kann man in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ geeignete Optimierungsphasen für eingebettete Sprachen zum Übersetzer hinzufügen. Dazu muß man die zur Optimie-

rung notwendige Information im AST, also auf der Ebene des Typsystems, sichtbar machen.

Hier liegen aber auch die Grenzen der Optimierung. Die im AST enthaltenen Werte kann man zur Übersetzungszeit nämlich nicht einsehen. Speichert man etwa Vektoren als Werte vom Typ `Vec3(int,int,int)` im AST, kann man zur Übersetzungszeit nicht feststellen, ob dessen Komponenten aus lauter Nullen bestehen.

Es lassen sich also nur solche Optimierungen realisieren, für die die im AST enthaltene Strukturinformation ausreicht. Da Optimierungen im Typsystem realisiert werden müssen, ergeben sich zusätzliche Beschränkungen, da die Typsprache von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ mangels Fixpunktkombinator nicht berechnungsuniversell ist. Optimierungsalgorithmen müssen sich wie Typsysteme auch allein durch Induktion über die Termstruktur von Termen implementieren lassen.

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ hat gegenüber System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ und System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ jedoch den entscheidenden Vorteil, daß die Typisierbarkeit entscheidbar ist und der partielle Auswerter terminiert. In der Praxis bedeutet das, daß ein Übersetzer terminiert.

Mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ wird die Beschränkung auf nicht-rekursive Sprachen aufgehoben, so daß sich eine größere Klasse von Sprachen integrieren läßt. Allerdings müssen Typsystem und Optimierungen sich auch hier allein durch strukturelle Induktion über den Termaufbau beschreiben lassen und es kann nicht mehr garantiert werden, daß der partielle Auswerter terminiert. Bei der konkreten Implementierung eines System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$-Übersetzers muß man daher geeignete Vorkehrungen treffen.

Erst mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ hat man auf der Typebene eine berechnungsuniverselle Sprache zur Hand. Dadurch lassen sich auch komplexere Typsysteme und Optimierungen realisieren, man verliert jedoch die Entscheidbarkeit der Typisierung.

Es empfiehlt sich zur Integration einer DSL möglichst auf die kleinste Sprache in der System $\mathbf{F}^{\mathrm{SA}}_{\omega}$-Hierarchie zurückzugreifen. Denkbar wäre z.B. einen Übersetzer so zu konzipieren, daß man die Sprachmerkmale von System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ und System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ explizit durch Optionen an den Übersetzer freischalten muß.

4.7 Verwandte Arbeiten:
Einbettung von DSLs in funktionalen Programmiersprachen

Die Einbettung von domänenspezifischen Sprachen in funktionale Sprachen ist besonders populär, weil diese mit ihren hochentwickelten Typsystemen und einer flexiblen Syntax eine kostengünstige Integration versprechen [73]. Beispiele für eingebettete DSLs finden sich z.B. in den Bereichen *Parsing* [110, 76, 130], *pretty printing* [75], Grafik [46, 43], Musik [72], Robotik [134], grafische Benutzeroberflächen [38], Datenbanken [100] und dem Hardware-Design [15, 109, 129].

Für die überwiegende Zahl dieser Beispiele wurde jedoch nicht wie in Abschnitt 4.2.2 am Beispiel von Haskell gezeigt, ein Interpreter realisiert, der die einzubettende Sprache als Zeichenkette übernimmt. Das Vorgehen entspricht vielmehr dem, was wir am Beispiel der Integration von **V** in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ gezeigt haben: Die domänenspezifische Sprache ergibt sich aus der Kombination von Funktionen der Hostsprache, wobei die Ausführung einer Funktion zu einem AST-Knoten führt. Wie in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ versucht man möglichst viele ungültige Terme mit dem Typsystem der Hostsprache herauszufiltern.

Genau hier liegt das Problem der meisten Ansätze mit standardisierten Sprachen. Der AST

wird durch einen algebraischen Datentyp dargestellt und Funktionen, die einen AST erweitern oder manipulieren, akzeptieren grundsätzlich jeden Wert dieses Typs, unabhängig davon, mit welchem Konstruktor er erzeugt wurde. Gerade bei der Integration von DSLs ist man aber häufig daran interessiert, Funktionen auf ausgesuchte Konstruktoren zu beschränken. Eine gängige Technik hierzu ist der Einsatz von *phantom types* [101]. Das sind algebraische Datentypen, bei denen ein oder mehr Typparameter auf der rechten Seite der Typdefinitionen keine Verwendung finden. Der eigentliche Verwendungszweck dient dem Informationstransport während der Übersetzungszeit (Abbildung 4.10 verdeutlicht das Prinzip).

```
data Unsafe = S String | B Bool

class Gen a where
  {- Der Argumenttyp wird
     ignoriert -}
  gen :: a -> Unsafe

instance (Show a) => Gen [a] where
  gen v = S (show v)
instance Gen Bool where
  gen b = B b



{- myAnd ist vom Typ -}
myAnd :: Unsafe -> Unsafe
         -> Unsafe
myAnd (B v1) (B v2) = gen (v1 && v2)

{- Ausführen von 'test' verursacht
   einen Laufzeitfehler. -}
test = (gen ''1'') `myAnd` (gen ''2'')
```

```
data Safe a = S String | B Bool

class Gen a where
  {- Der Argumenttyp wandert in das
     Phantomtyp-Argument -}
  gen :: a -> Safe a

instance (Show a) => Gen [a] where
  gen v = S (show v)
instance GenConst Bool where
  gen b = B b

{- Die folgende Typannotation schränkt
   die Funktion myAnd auf den Typ
   'Safe Bool' ein. -}
myAnd :: Safe Bool -> Safe Bool
         -> Safe Bool
myAnd (B v1) (B v2) = gen (v1 && v2)

{- Übersetzen von 'test' ergibt einen
   Typfehler, da   gen ''1'' :: Safe a
   nicht zum Typ   Safe Bool   paßt.
-}
test = (gen ''1'') `myAnd` (gen ''2'')
```

Abbildung 4.10: Einschränkung der Funktion `myAnd` auf bestimmte Konstrukorwerte mit Hilfe von *phantom types*. Wird die Funktion `test` im linken Haskell-Code-Fragment aufgerufen, so kommt es zu einem Laufzeitfehler, da die Funktion `myAdd` keinen Behandlungsfall für mit dem Konstruktor `S` generierte Werte bereithält. Die rechte Seite der Abbildung zeigt den Einsatz von *phantom types*: Die überladene Funktion `gen` initialisiert den Phantom-Typparameter so, daß man ihm den Typ des gespeicherten Werts entnehmen kann. Ein mit `gen True` erzeugter Wert hat dann den Typ `Safe Bool`. Durch die explizite Vorgabe des Funktionstyps von `myAnd`, wird der Definitionsbereich dieser Funktion entsprechend eingeschränkt und die Funktion `test` wird vom Typechecker als fehlerhaft zurückgewiesen.

Wie Hallgren in [63] zeigt, kann man Instanzdefinitionen einer Typklasse in Haskell als Prädikate über Typen, oder (im Fall von Multiparameter-Typklassen), als Relationen zwischen Typen auffassen und so einfache logische Programme allein auf der Typebene (unter Ausnutzung des Typecheckers) ausführen lassen.

Thiemann demonstriert in [164] am Beispiel einer eingebetteten DSL zur Generation von XHTML, daß *phantom types* in Kombination mit dieser Type-Level-Sprache die Integration von DSLs mit komplexem Typsystem erlauben.

In [163] kommt er jedoch zu dem Schluß, daß der dabei entstehende Programmcode schwer lesbar und wartbar ist. Als einen Grund führt er den Paradigmenwechsel vom funktionalen Paradigma (Haskell) zum logischen Paradigma (Typebene) an. Als Ausweg schlägt er vor, zur Typüberprüfung von eingebetteten DSLs auf das von ihm, Gasbichler, Neubauer und Sperber entwickelte funktional-logische Überladen (engl.: *functional logic overloading*) zurückzugreifen [122]. Bei dieser Technik kann Überladung durch ein Termersetzungssystem gesteuert werden, welches auf der Ebene des Typsystems codiert wird – Typklassen übernehmen dann die Rolle von Typfunktionen.

Das Termersetzungssystem wird als funktional-logisches Programm ausgeführt, wobei man für jede Typfunktion eine eigene Reduktionsstrategie vorgeben kann (z.B. kann man vereinbaren, daß jeweils die im Quelltext zuerst vereinbarte Reduktionsregel oder aber die am besten passende Reduktionsregel greift). Im Vergleich zum *pattern matching* mit `Typecase` ist man damit zwar nicht flexibler, kann Programme auf der Typebene in manchen Fällen aber lesbarer formulieren, da man die jeweilige Strategie nicht explizit codieren muß.

Die Terminierung des Termersetzungsystems ist nicht garantiert, woraus folgt, daß die Typisierbarkeit eines Terms nicht mehr entscheidbar ist. In unserem Modell geht die Entscheidbarkeit der Typisierung erst mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ verloren. Viele DSLs lassen sich jedoch bereits in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ oder in System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ codieren, womit die Entscheidbarkeit der Typisierung erhalten bleibt. Im Falle von System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ kann jedoch nicht garantiert werden, daß der partielle Auswerter terminiert.

System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ ist sowohl auf der Typ-, als auch auf der Termebene eine rein funktionale Sprache, was sicherlich ein Vorteil ist. Darüber hinaus hat das Programmieren mit Typtermen in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ einen expliziteren Charakter, da der Programmierer Typberechnungen direkt anstoßen kann. In Haskell hingegen werden Typberechnungen während der Inferenz von Typinformation angestoßen und erscheinen dem Programmierer daher implizit, was das Aufspüren von Fehlern u.U. schwieriger gestaltet.

Template Haskell [151] ist eine Erweiterung von Haskell zu einer zweistufigen Sprache. Im Unterschied zu den gerade besprochenen Spracherweiterungen, ist aber keine der Stufen ins Typsystem eingegossen. Ähnlich wie in MetaOCaml kann der Programmierer die Auswertung von Programmcode durch spezielle Quotierungen zurückstellen. Das Ergebnis von quotiertem Code ist jedoch kein Bytecode, sondern der AST des Codes, den man mit ganz normalen Haskell-Funktionen manipulieren kann. Programmcode, der mit dem *splice*-Operator markiert ist, wird zur Compilezeit ausgeführt. Dabei kann es sich um beliebigen Haskell-Code handeln, der z.B. auch Ein-/Ausgabe-Operationen durchführen darf. Das Ergebnis der Programmausführung muß ein gültiger AST sein, der dann durch *unparsing* in Haskell-Quellcode zurücktransformiert wird und an die Stelle des *splice*-Aufrufs geschrieben wird. Der *splice*-Operator ist daher eine Mischung aus dem *run*- und dem *escape*-Operator in MetaOCaml.

DSL-Code wird durch Quotieren während der Übersetzungszeit zugänglich und kann mit normalen Haskell-Funktionen bearbeitet werden. Zur Typüberprüfung muß man daher nicht auf komplizierte Typfunktionen zurückgreifen, sondern kann direkt auf dem AST operieren. Der freie Zugriff auf den AST der DSL ermöglicht darüber hinaus die Einbettung von domänenspezifischen Optimierungen. Seefried, Chakravarty und Keller demonstrieren diese Technik in [149].

4.8 Weiterführende Literatur

Die Einbettung von Sprachen durch strukturelle Typanalyse ist ein neues Feld, zu dem es bisher keine Literatur gibt. Auf Literatur zu verwandten Arbeiten haben wir im vorangegangen Abschnitt bereits hingewiesen.

Die Idee der partiellen Auswertung von Interpretern geht wahrscheinlich zurück auf Futamura [51]. Jones gibt in [86] viele Empfehlungen, die dabei helfen sollen, einen Interpreter so zu implementieren, daß ein partieller Auswerter einen Großteil des Interpretationsaufwandes (z.B. *cases analysis*) übernimmt. Glenstrup und Jones stellen in [56] einen Algorithmus zur Binding-Time-Analyse vor, der garantiert terminiert und demonstrieren seine Anwendung an diversen Interpretern.

Bereits Lisp bringt mit *Quoting*, *Unquoting* und der Funktion `eval` alle Eigenschaften einer Multi-Stage Sprache mit [41], allerdings dürfen Quotes im Verlgeich zu MetaOCaml nicht verschachtelt werden. Ein in Scheme (einem Lisp-Derivat) implementierter *staged interpreter* findet sich z.B. bei Abelson, Sussman und Sussman [6].

Die Konzepte von MetaOCaml, sowie ein theoretisches Modell für Multi-Stage Programmiersprachen und Anwendungsbeispiele, findet man in der Dissertation von Taha [159].

Czarnecki, O'Donnel, Striegnitz und Taha vergleichen in [40] die Sprachen Template Haskell, MetaOCaml und C++ bezüglich ihrer Eigenschaften und Möglichkeiten zur Spracheinbettung.

Kapitel 5

C++ Template-Metaprogrammierung

Inkarnation von System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ in C++

Programmiersprachen mit Unterstützung zur strukturellen Typanalyse sind rar und werden vornehmlich als Zwischensprache eingesetzt, in die eine Hochsprache übersetzt wird, um anschließend typgesteuerte Optimierungen vorzunehmen (zum Beispiel werden Varianten von λ_i^{ML} als Zwischensprache des FLINT/ML- [150] und des TIL/ML-Compilers [162, 117] eingesetzt).

Eine der wenigen Programmiersprachem des *mainstream*, die es erlaubt mit struktureller Typanalyse zu experimentieren, ist C++ . Dies scheint im ersten Moment erstaunlich, da sich weder im C++ -Standard noch in etablierten Lehrbüchern zu C++ entsprechende Hinweise finden. Beim Design von C++ hatte man die entsprechenden Programmiertechniken (Template-Metaprogrammierung) auch gar nicht im Auge. Sie wurden vielmehr rein zufällig entdeckt [167, 166].

In diesem Kapitel wollen wir die Brücke zwischen Theorie zur Praxis schlagen und zeigen, wie sich diverse Sprachkonzepte von System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ in C++ abbilden lassen.

In den ersten beiden Abschnitten werden wir die für uns zentralen Eigenschaften von C++ kurz vorstellen. Aus Platzgründen müssen wir uns dabei auf das Wesentliche beschränken. Dem im Umgang mit C++ unerfahrenen Leser wird daher dringend die Lektüre von einführender Literatur empfohlen (z.B. [158, 116] als generelle Einführung oder [168, 156] zur Vertiefung von C++ -Templates).

Im dritten Abschnitt wird anhand von Beispielen gezeigt, wie sich Konzepte aus System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ in C++ simulieren lassen, welche Unterschiede es gibt und wo man Kompromisse eingehen muß. Abschließend werden wir die Programmierung mit struktureller Typanalyse in C++ kritisch bewerten.

5.1 Überladung in C++

C++ erlaubt das Überladen von Funktionen und Methoden. Das heißt, ein Funktions- oder Methodenbezeichner kann mehreren Definitionen zugeordnet sein. Voraussetzung ist, daß sich die einzelnen Definitionen im Typ und/oder der Zahl der Argumente unterscheiden.

Betrachten wir ein einfaches Beispiel:

```
int f(int x,int y) { return x + y; }
double f(double x) { return -x;    }
```

Im obigen Codefragment wurden zwei Definitionen zum Funktionsbezeichner `f` vereinbart. Die erste Version erwartet als Argument zwei Werte vom Typ `int` und liefert einen `int`-Wert als Ergebnis zurück. Die zweite Version erwartet und generiert einen Wert vom Typ `double`.

Die Zuordnung eines Funktionsaufrufs zu einer Funktionsdefinition (man spricht vom Auflösen von Überladung) trifft der C++ -Übersetzer allein auf Grundlage der Argumenttypen. Das Überladen von Funktionen, die sich nur im Ergebnistyp unterscheiden, ist nicht erlaubt.

Diese Einschränkung ist sinnvoll, denn würde man das Hinzunehmen der Funktionsdefinition `int f(double x)` erlauben, könnte man in der Anweisung

```
cout << f(3) << endl;
```

nicht entscheiden, auf welche Definition von `f` Bezug genommen wird (`int f(double)` oder `double f(double)`).

Neben der Überladung von Funktionen, erlaubt C++ das Überladen von Operatoren. Dazu gehören z.B. arithmetische- , boolesche- und shift-Operatoren, aber auch der Vergleichs- und der Sequenzoperator (Komma-Operator).

Operatorüberladung wird intern auf Funktionsüberladung zurückgeführt, indem jedem Operatorsymbol ein spezielles Funktionssymbol zugeordnet wird. Zum Beispiel heißt die zum Additionsoperator gehörende Funktion `operator+`. Das Überladen eines Operators entspricht dann dem Überladen der zugehörigen Funktion.

Tatsächlich sind infix-, prefix- und postfix-Operatoren in C++ nur syntaktischer Zucker, denn intern werden alle Operatorapplikationen auf Funktionsaufrufe abgebildet. So entsteht zum Beispiel aus dem Term

```
a + b * c * d
```

der Term

```
operator+(a, operator*(operator*(b,c),d))
```

Die zugehörige Abbildung ist eindeutig, da Operatorpräzedenzen und Assoziativitäten im C++ -Standard fest vorgegeben sind.

Weitere Möglichkeiten zur Überladung ergeben sich im Zusammenhang mit Klassen und Objekten. Durch das Überladen spezieller Methoden kann man z.B. festlegen, wie ein Objekt auf die Anwendung des Indizieroperators (`operator[]`) reagieren soll, so daß es sich dem Programmierer z.B. wie ein Datenfeld darstellt. Durch Überladen der Klammeroperator-Methode (Funktionsmethode, `operator()`) kann sich ein Objekt wie eine Funktion verhalten. Überlädt man den Zuweisungsoperator (`operator=`), kann man auf die Zuweisung von Werten an ein Objekt reagieren, und um festzulegen, daß sich ein Objekt verhalten kann, als hätte es einen anderen Typ, kann man spezielle Konvertierungsoperatoren überladen.

Die Anwendung eines Operators auf ein Objekt wird intern auf einen Methodenaufruf abgebildet. Abbildung 5.1 zeigt als Beispiel die Klasse `Ovl`, in der alle oben genannten Operatoren

```
#include <string>
class Ovl {
  std::string sv;
public:
  // Indizieroperatoren
  int operator[](int i);
  int operator[](std::string s);
  // Funktionssimulationsoperatoren
  int operator()(int a,int b);
  std::string operator()(double x,int z,std::string s);
  // Zuweisungsoperator
  Ovl& operator=(std::string s) {
    sv = s;
    return *this;
  }
  // Konvertierungsoperator
  operator std::string() { return sv; }
};

int main() {
  Ovl o;
  std::string s;
  o[2];                // o.operator[](2)
  o["Hello"];          // o.operator[]("Hello")

  o(3,4);              // o.operator()(3,4)
  o(2.34,4, "World"); // o.operator()(2.34, 4, "World")

  o="Test";            // o.operator=("Test")
  s = o;               // s.operator=(o.string())
}
```

Abbildung 5.1: Beispiel zur Operatorüberladung durch Definition geeigneter Methoden.

überladen wurden. Die Abbildung auf einen Methodenaufruf haben wir jeweils im Kommentar hinter den jeweiligen Operatoraufrufen vermerkt.

Der Aufruf von Konvertierungsoperatoren erfolgt implizit; d.h. man kann ihn im Gegensatz zu den anderen Operatoraufrufen nicht direkt aus dem Quelltext ablesen. Um die Zuweisung eines Objekts vom Typ `Ovl` an eine Zeichenkette `s` zu ermöglichen, muß entweder ein passender Zuweisungsoperator in der `string`-Klasse definiert sein, oder die Klasse `Ovl` stellt einen Konvertierungsoperator bereit, der ein `Ovl`-Objekt in ein `string`-Objekt konvertiert. In unserem Beispiel trifft die zweite Bedingung zu und der Übersetzer ergänzt automatisch einen Aufruf des Konvertierungsoperators `operator std::string`[1].

Das Auflösen von Überladung ist in C++ ein sehr komplexer Vorgang, da man bei der Suche nach einer passenden Funktionsdefinition auf eingebaute und benutzerdefinierte Konvertierungen Rücksicht nehmen muß. Der Einfachheit halber, lassen wir Konvertierungen im Rest der

[1]Ein Ergebnistyp muß für Konvertierungsoperatoren nicht angegeben werden, da er sich direkt aus dem Operatornamen ergibt.

Arbeit weitgehend außer acht.

Bei den meisten C++ -Übersetzern ist das Auflösen von Überladung ein rein syntaktischer Vorgang[2]: In einem Vorverarbeitungsschritt werden Funktionen und Methoden so umbenannt, daß sich ihr Typ in ihrem Namen wiederspiegelt (z.B. könnte der Funktionsbezeichner `f` in der Definition `double f(int)` in `f_int__double` umbenannt werden). Funktions- und Methodenaufrufe werden entsprechend umbenannt, so daß es am Ende nur noch eindeutige Bezeichnernamen gibt. Den Umbenennungsprozeß nennt man ***name mangling*** und er ist leider kein Teil des C++ -Standards, was den Austausch von C++ -Bibliotheken erschwert.

Für uns ist wichtig, daß Überladung in C++ , wie in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ein Übersetzungszeitmechanismus ist.

5.2 C++ -Templates

5.2.1 Klassentemplates

C++ -Templates wurden primär zur effizienten Programmierung mit generischen Datentypen entwickelt. Zur Motivation betrachten wir eine einfache Stackklasse:

```
class intStack {
public:
  int* theStack;
  int stackSize;
  int t;

  intStack( int s )   { theStack = new int[s]; t=0; stackSize=s; }
  ~intStack()         { delete [] theStack; }
  bool isEmpty()      { return t==0; }
  int top()           { return theStack[t-1];   }
  int pop()           { return theStack[- -t]; }
  intStack& push(const int& e){ theStack[t++] = e;
                                return *this;
                              }
};
```

`intStack` ist ein abstrakter Datentyp (eine Klasse) zur Modellierung von Stapelspeichern (engl. *stacks*), auf denen Werte vom Typ `int` abgelegt werden.

Der eigentliche Stack wird durch das Datenfeld `theStack` repräsentiert; die Stackspitze `t` ist ein Index in dieses Feld und in `stackSize` ist die Kapazität des Stacks, also die Zahl der Elemente, die auf dem Stack insgesamt abgelegt werden können, hinterlegt.

Der Konstruktor übernimmt die Stackkapazität, reserviert entsprechend Speicher für das Datenfeld und initialisiert den Stackzeiger auf Null. Im Destruktor (`~intStack`) wird der vom Datenfeld belegte Speicher wieder freigegeben.

Die Stack-Funktionalität verbirgt sich in den Methoden `isEmpty`, `top`, `push` und `pop`. Mit `isEmpty` kann geprüft werden, ob ein Stack leer ist. `top` erlaubt die Abfrage des obersten

[2]Der C++ -Standard selbst macht keine Vorgaben zur Implementierung von Überladung.

Stack-Elementes, ohne dieses zu löschen. `push` legt ein neues Objekt auf dem Stack ab und gibt eine Referenz auf den Stack als Ergebnis zurück und schließlich dient die Methode `pop` dem Lesen und gleichzeitigen Entfernen des obersten Stackelements.

Stacks zur Aufnahme von Werten eines anderen Typs lassen sich ähnlich implementieren. Entsprechende Klassen lassen sich leicht aus `intStack` ableiten, indem man in der Klassendefinition überall dort, wo sich ein Typ auf den Elementtyp bezieht, den neuen Typ einsetzt. Eine doppelte Codierung der Stackfunktionalität bedeutet aber einen Verstoß gegen das Abstraktionsprinzip.

Dieser Situation sind wir bereits bei der Motivation von System **F** begegnet (siehe Seite 39). Um redundanten Code zu vermeiden, hatten wir die Funktion `twice`, die eine Funktion zweimal auf ein Argument anwendet, mit einem Typparameter versehen:

$$\texttt{twice} = \Lambda\alpha :: *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x))$$

C++ erlaubt einen ähnlichen Mechanismus für Klassen, Methoden und Funktionen. Klassendefinitionen können über einen oder mehrere Typen parametrisiert werden, wobei die Typparameter in einer speziellen Template-Deklarationsliste der Klassendeklaration vorangestellt werden. Innerhalb der Klassendeklaration dürfen Typparameter dann überall dort eingesetzt werden, wo normalerweise ein C++ -Datentyp erwartet wird.

```
template <typename T>
class Stack {
public:
  T* theStack;
  int stackSize;
  int t;

  Stack( int s ) { theStack = new T[s]; t=0; stackSize=s; }
  ~Stack()       { delete [] theStack; }
  bool isEmpty() { return t==0; }
  T top()        { return theStack[t-1];   }
  T pop()        { return theStack[- -t]; }
  Stack<T>& push(const T& e){ theStack[t++] = e;
                              return *this;
                            }
};
```

Abbildung 5.2: Generischer Stack in C++ .

Abbildung 5.2 zeigt eine generische Version der Stackklasse. In der `template`-Deklaration wird der Typparameter `T` eingeführt. Im Deklarationsteil taucht er an allen jenen Stellen auf, wo in der Klasse `intStack` der Elementtyp genannt wurde.

`Stack` ist jedoch kein Datentyp! Klassentemplates sind vielmehr Schablonen (engl.: *Templates*) zur **Erzeugung von Typen**.

Einen konkreten Typ (eine Instanz) gewinnt man, indem man hinter dem Namen des Klassentemplate für jeden Typparameter ein Typargument angibt, wobei die Liste der Typparameter

vom Klammerpaar < und > umschlossen sein muß. Zum Beispiel generiert `Stack<int>` einen Stacktyp zur Aufnahme von `int`-Werten:

```
Stack<int> intStack(10);
intStack.push(10);
intStack.push(20);
```

Trifft der C++ -Übersetzer auf die Instanziierung `Stack<int>`, so wird das `Stack`-Template expandiert; d.h. es wird ein neuer Klassentyp generiert, indem der Typparameter `T` an den Typ `int` gebunden wird, und im Rumpf der Template-Klasse `Stack` alle Vorkommen von `T` durch `int` ersetzt werden. Ähnlich dem *name mangling* bei der Auflösung von Überladung, werden der Name der neu entstandenen Klasse, sowie die Namen ihrer Konstruktoren und Destruktoren geändert und der restliche Quelltext entsprechend angepaßt.

Abbildung 5.3 zeigt, wie das Ergebnis der Template-Expansion von `Stack` aussehen könnte.

```
class _int__Stack {
public:
  int* theStack;
  int stackSize;
  int t;

  _int__Stack( int s ) { theStack = new int[s]; t=0 ; stackSize=s; }
  bool isEmpty()       { return t==0;  }
  int top()            { return theStack[t-1];   }
  int pop()            { return theStack[- -t]; }
  _int__Stack& push(const int& e){ theStack[t++] = e;
                                   return *this;
                                 }
};
//...
_int__Stack intStack(10);
intStack.push( 10 );
intStack.push( 20 );
```

Abbildung 5.3: Durch Template-Expansion entstandene Stack-Klasse. Der Klassenname wird vom Übersetzer automatisch gewählt und der Quelltext entsprechend angepaßt.

Nach der Expansion wird die neu entstandene Klasse auf Typkonsistenz hin untersucht – dabei kann es unter Umständen zu weiteren Template-Expansionen kommen.

Der C++ -Standard fordert im wesentlichen die syntaktische Korrektheit von Templates. Ein Test von Template-Code auf Typsicherheit ist nicht vorgeschrieben. Es ist auch nicht garantiert, daß jede mögliche Instanz eines Template zu typsicherem Code führt (im Vergleich z.B. zu System **F** oder konkreten Programmiersprachen wie PIZZA [128], Haskell oder ML).

Im Template-Code können implizit Bedingungen an die Typparameter verborgen sein. Das Erkennen dieser Bedingungen ist nicht immer einfach und erfordert ein wenig Erfahrung im Umgang mit C++ . Zum Beispiel impliziert das `Stack`-Template, daß der Typ `T`, mit dem der Stack expandiert wird, einen default-Konstruktor aufweisen muß. Ansonsten kann das Feld `theStack` nämlich nicht via `new T[s]` initialisiert werden. Da die Methoden `pop` und `top` ihre

Funktionsergebnisse *per value* zurückgeben, muß `T` darüber hinaus einen Kopierkonstruktor bereitstellen.

Wir wollen noch ein Beispiel betrachten, bei dem die Bedingungen an ein Typargument etwas offensichtlicher sind und zeigen, was man tun kann, wenn eine Bedingung für ein bestimmtes Typargument nicht erfüllt ist.

Angenommen, man möchte zwei Stacks miteinander vergleichen. Dann könnte man z.B. die folgende Methode zum Klassentemplate `Stack` hinzunehmen:

```
bool Stack<T>::cmpStack(const Stack<T>& other) {
  int p = t;
  if (t != other.t) return false;
  else {
    while ( -p >= 0 ) {
      if (! (theStack[p] == other.theStack[p]))
        return false;
    }
  }
  return true;
}
```

Zwei Stacks sind ungleich, wenn sie unterschiedlich viele Elemente speichern. Speichern sie gleichviele Elemente, so sind zwei Stacks genau dann gleich, wenn an entsprechenden Stackpositionen identische Elementwerte abgelegt sind. Da der Vergleich durch Anwenden von `operator==` vollzogen wird, kann diese Methode nur dann angewendet werden, wenn das Stacktemplate mit einem Typ instanziiert wurde, für den eine passende Überladung von `operator==` zur Verfügung steht.

Diese Bedingung kommt aber erst dann zum tragen, wenn die Methode `cmpStack` verwendet wird. Betrachten wir dazu das Codebeispiel aus Abbildung 5.4.

Für den Typ `intVec` ist der Vergleichsoperator nicht definiert. Dennoch kann das um `cmpStack` erweiterte Stacktemplate mit `intVec` instanziiert werden, ohne daß es zu einem Typfehler kommt. Nicht-virtuelle und statische Methoden werden nämlich erst dann expandiert, wenn deren Definition gebraucht wird (sie also aufgerufen werden). Dieses Konzept heißt **verzögerte Expansion** (engl.: *lazy instantiation* - siehe [168, Seite 143]).

Verzögerte Expansion gewährt einen großzügigeren Umgang mit den impliziten Randbedingungen an einen Typparameter, da die Einhaltung einer Bedingung, die für einen bestimmten Anwendungsfall bedeutungslos ist, nicht erzwungen wird. C++ -Templates sind dadurch zwar flexibler einsetzbar, jedoch hat dieses Konzept auch Nachteile.

Kommt es in Folge einer Template-Expansion zu einem Typfehler, so bezieht sich die Fehlermeldung auf den Quelltext des zugehörigen Templates - im obigen Beispiel wird sie einen Verweis auf die Programmzeile enthalten, in der der fehlende

`operator==:intVec×intVec→bool`

aufgerufen wird.

Um die Fehlermeldung zu verstehen, muß der Programmierer sich mit der Implementierung der `Stack`-Klasse auseinandersetzen. In Bezug auf Komponenten und Bibliotheken ist das

```
class intVec {
public:
  int x1; int x2 ; int x3;
  intVec(int _x1=0, int _x2=0, int _x3=0) {
    x1=_x1 ; x2=_x2 ; x3=_x3;
  }
};

Stack<intVec> a(10);  a.push( intVec(1,2,3) );
Stack<intVec> b(10);  b.push( intVec(1,2,3) );

// ... bis hier okay!
if (a.cmpStack(b))        // <-- Typfehler!
  cout << "Yes." << endl; // operator==(intVec,intVec)
                          // nicht definiert.
```

Abbildung 5.4: Verzögerte Template-Expansion erlaubt das Arbeiten mit Datentypen, die nur partiell typsicher sind. Aber nur solange sich die Verwendung auf typsicheren Programmcode beschränkt.

natürlich absolut kontraproduktiv - man will ja von deren Implementierung profitieren und sie nicht im Detail verstehen müssen.

Ist ein Template für eine bestimmte Typargument-Kombination nicht benutzbar, so bietet C++ grundsätzlich zwei Auswege. Gehen wir exemplarisch davon aus, daß ein generischer Typ `C` von einem Typparameter abhängt und die Instanziierung mit `A` zu einem Typfehler führt. Um `C` mit `A` verwendbar zu machen könnte man:

1. nachträglich dafür sorgen, daß alle Randbedingungen erfüllt sind (d.h. man müßte die Implementierung von `A` ggf. anpassen und/oder fehlende Funktionen, Operatoren oder Datentypen zur Verfügung stellen); oder

2. das Klassentemplate spezialisieren; d.h. man ordnet der Instanz `C<A>` eine spezielle Klassendefinition mit einer geänderten Semantik zu.

Übertragen auf unser `Stack`-Beispiel stellt man also entweder eine geeignete Überladung von `operator==` bereit, oder man definiert eine Spezialisierung des Stack-Templates, die eine angepaßte Version der `cmpStack`-Methode bereithält (siehe Abbildung 5.5).

Abgesehen von der modifizierten `cmpStack` Methode, entspricht diese Klasse exakt dem, was der Übersetzer automatisch aus der Klasse `Stack` generiert hätte[3].

Trifft der Übersetzer jetzt auf die Instanz `Stack<intVec>`, so wird nicht das allgemeine Klassentemplate

```
template <class T>
class Stack {...}
```

[3]Es sei erwähnt, daß der C++ -Standard in dieser Situation auch das gezielte Spezialisieren der Methode `cmpStack` erlaubt.

```
template <>
class Stack<intVec> {
public:
  intVec* theStack;
  int stackSize;
  int t;

  Stack( int size )           { theStack = new intVec[size]; t=0;
                                stackSize=size; }
  bool isEmpty()              { return t==0; }
  intVec top()                { return theStack[t-1];  }
  intVec pop()                { return theStack[- -t];}
  Stack& push(const intVec& e){ theStack[t++] = e;
                                return *this;
                              }
  bool cmpStack(const Stack& other) {
    int p = t;
    if (t != other.t) return false;
    else { while ( -p >= 0 ) {
             if (!((theStack[p].x1 == other.theStack[p].x1) &&
                   (theStack[p].x2 == other.theStack[p].x2) &&
                   (theStack[p].x3 == other.theStack[p].x3)) )
               return false;
           }
    }
    return true;
  }
};
```

Abbildung 5.5: Für den Typ `intVec` spezialisiertes Klassentemplate `Stack`. Der Vergleich zweier Vektoren erfolgt jetzt komponentenweise. Für den eingebauten Typ `int` ist der Vergleichsoperator vordefiniert.

(das auch **Mutter-Template** genannt wird), sondern das spezialisierte Klassentemplate

```
template <> class Stack<intVec>
{...}
```

expandiert.

Spezialisierte Klassentemplates müssen in keinerlei Beziehung zueinander stehen. Sie dürfen sich sowohl hinsichtlich ihrer Schnittstelle, als auch hinsichtlich ihrer Implementierung vollständig voneinander unterscheiden. Das einzige, was ihnen gemein ist, ist der Kopf im Typterm (hier: `Stack`).

Die Möglichkeiten zur Spezialisierung von Templates sind sehr ausgeprägt und mächtig. Der Programmierer kann Templates nämlich auch partiell spezialisieren:

```
// Mutter-Template:
template <typename T1,typename T2>  class Foo  {  };
//  class Foo<T1,T2> { ... };              /* 0 */

//Spezialisierungen:
template <>
class Foo<int,double> { };                /* 1 */

template <typename T1>
class Foo<Stack<int>,T1> {};               /* 2 */

template <typename T1>
class Foo<T1,Stack<int> > { };             /* 3 */
```

Bei der Spezialisierung eines Klassentemplates befindet sich hinter dem Schlüsselwort `class` ein Typterm, dessen Kopf dem zu spezialisierenden Template entspricht (hier `Foo`) und alle in der `template`-Deklaration eingeführten Typvariablen enthalten darf. Diesen Typterm nennt man *Muster* (engl. *pattern*).

Um nicht weiter zwischen Mutter-Template und Spezialisierung unterscheiden zu müssen, tun wir im Folgenden so, als ob Mutter-Templates auch in folgender Form angegeben werden dürften:

```
// Kein \Cpp{} !
template <typename T1,typename T2>  class Foo<T1,T2>  { };
```

Das erlaubt uns die einfachere Anschauung, daß *jede* Klassentemplate-Deklaration aus einer Deklaration von Typvariablen (`template`), einem Muster (`Foo<T1,T2>`) und einem Klassenkörper (`{...};`) besteht.

Um einem variablenfreien Typterm (also einem Typ) eine Klassentemplate-Deklaration zuzuordnen, geht der C++ -Übersetzer so vor, daß er zunächst alle diejenigen Template Spezialisierungen heraussucht, deren Muster sich mit dem konkreten Typterm unifizieren lassen (*pattern matching*).

Führt das *pattern matching* zu mehreren Lösungen, so wird versucht, die „speziellste" Deklaration zu identifizieren. Ist das nicht möglich, kommt es zu einem Doppeldeutigkeitsfehler (engl. *ambiguity error*).

Zur Bestimmung der speziellsten Deklaration definiert der C++ -Standard eine Halbordnung über den Typtermen (den Mustern der Klassentemplates), die im wesentlichen der Matchordnung, die wir auf Seite 27 für Terme eingeführt haben, entspricht.

Zum Beispiel bezieht sich der Typ `Foo<Stack<int>, int>` auf die zweite Spezialisierung, denn mit der Substitution

$$subst = \{\texttt{T1} \mapsto \texttt{Stack<int>}\}$$

gilt: `Foo<T1,T2>` $\trianglelefteq$ `Foo<Stack<int>,T2>`.

Für den Typ `Foo<Stack<int>,Stack<int> >` kommen sowohl das Muster des Muttertemplates, als auch die Muster `Foo<Stack<int>,T1>` und `Foo<T1,Stack<int> >` in Frage. Die

letzgenannten Muster sind spezieller als das Muttertemplate, denn es gilt jeweils mit $subst = \{\texttt{T1} \mapsto \texttt{Stack<int>}\}$:

$$\texttt{Foo<T1,Stack<int> >} \trianglelefteq \texttt{Foo<Stack<int>,Stack<int> >}$$
$$\texttt{Foo<Stack<int>,T1>} \trianglelefteq \texttt{Foo<Stack<int>,Stack<int> >}$$

Wegen

$$\texttt{Foo<Stack<int>,T1>} \equiv \texttt{Foo<T1,Stack<int> >}$$

läßt sich jedoch kein allgemeinstes Muster bestimmen und es kommt zu einem Doppeldeutigkeitsfehler.

5.2.2 Funktions- und Methodentemplates

Wie Klassentemplates, werden Funktions- und Methodentemplates durch die Deklaration von Typparametern eingeleitet, die in der sich anschließenden Funktions-/Methodendefinition überall dort eingesetzt werden können, wo normalerweise ein C++ -Datentyp erwartet wird.

Eine generische Funktion zur Ausgabe eines beliebigen Stacks auf den Bildschirm könnte z.B. durch folgendes Funktionstemplate implementiert werden:

```
template <class T>
void printStack(const Stack<T>& stack) {
  for (int i=0 ; i<stack.stackSize ; ++i)
    std::cout << stack.theStack[i] << std::endl;
}
```

Ein Funktionstemplate ist keine C++ -Funktion: Es ist zum Beispiel nicht möglich die Adresse eines Funktionstemplates zu berechnen. Aus Benutzersicht verhält es sich jedoch wie eine gewöhnliche C++ -Funktion, da es mit geeigneten Argumenten ganz normal aufgerufen werden kann:

```
Stack<double>  doubleStack(10);
Stack<int>     intStack(10);

printStack(doubleStack);
printStack(intStack);
```

Der C++ -Übersetzer versucht die Argumenttypen des Funktionsaufrufs mit den Argumenttypen des Funktionstemplates zu unifizieren. Ist das möglich, kann man aus dem Unifikator sofort auf die Typen schließen, mit denen das Funktionstemplate instanziiert werden muß. Das Funktionstemplate wird expandiert, wobei eine neue Funktionsdefinition entsteht, der mittels *name mangling* ein eindeutiger Name zugeordnet wird.

Folglich beziehen sich die Aufrufe von `printStack` im obigen Beispiel auf unterschiedlichen Programmcode. Im ersten Fall wird das `printStack`-Template mit dem Typ `double` instanziiert, im zweiten mit dem Typ `int`.

Analog zu Klassentemplates findet eine Typüberprüfung von Funktionstemplates erst nach der Template-Expansion statt und auch Funktionstemplates können implizite Bedingungen an Typargumente enthalten.

Die Definition von Methodentemplates verläuft analog zu der Definition von Funktionstemplates.

5.2.3 C++ -Templates im Vergleich zu System F und System $\mathbf{F}^{\mathbf{SA}}_{\omega,1}$

Ein selbst bei Experten weit verbreitetes Mißverständnis ist die Gleichsetzung von C++ -Templates mit parametrischer Polymorphie, so wie wir sie für System **F** beschrieben haben und wie sie z.B. auch von Cardelli und Wegner in [30] beschrieben wird.

Der C++ -Template-Mechanismus ist jedoch orthogonal zu parametrischer Polymorphie. In System **F** ist polymorphen Funktionen ein Typ zugeordnet und das Typsystem garantiert, daß jede Typapplikation zu einer typsicheren Funktion führt. Wie wir gesehen haben, gilt das für C++ -Templates im Allgemeinen nicht.

Parametrisch-polymorphe Funktionen sind in System **F** wirklich universell, jede Instanz wird aus ein und demselben Stück Quelltext gewonnen und führt bei homogener Übersetzungsstrategie zu einem Stück Objektcode.

Es gibt aber auch Varianten von System **F** bei denen man die Zahl der gültigen Typargumente eingrenzen kann (man spricht dann von beschränkter parametrischer Polymorphie bzw. *bounded polymorphism*).

Zum Beispiel definiert System $\mathbf{F}_{<:}$ [28], eine um Subtyping erweiterte Variante von System **F**, eine Halbordnung $<:$ über Typen und man kann fordern, daß der Typ eines Funktionsargumentes bzgl. $<:$ kleiner sein muß als eine bestimmte Obergrenze. Diese Einschränkung wird dann im Funktionstyp sichtbar (z.B. $\forall\alpha <: T.\texttt{int}$, falls nur Argumente mit Typ $<: T$ akzeptiert werden).

Ähnlich kann man in Haskell verlangen, daß ein Typargument einer bestimmten Typklasse angehören muß [62], was sich ebenfalls im Typ einer Funktion niederschlägt (z.B. beschreibt `Eq a=> a -> a` eine polymorphe Funktion, die nur für Typen instanziiert werden kann, die Mitglied der Typklasse `Eq` sind).

In beiden Fällen sind die Bedingungen an ein Typargument expliziter Bestandteil des Typs der polymorphen Funktion und müssen nicht mühsam aus dem Quelltext inferiert werden. Es genügt also, die Interface-Spezifikation einer Funktion oder eines Datentyps zu kennen, um zu prüfen, ob eine Instanz zu typsicherem Programmcode führt.

Parametrische Polymorphie (ob beschränkt oder nicht), ist immer mit der Garantie verbunden, daß der Programmierer ausschließlich typsichere Instanzen generieren kann (diese Eigenschaft bildet schließlich auch den Schlüssel für eine separate Übersetzung von generischem Programmcode).

C++ -Templates erfüllen diese Bedingung nicht! Das C++ -Typsystem kennt keine polymorphen Typen. Ein Template scheint zwar polymorph, hat aber keinen Typ und wird auch keiner Typüberprüfung unterzogen. Templates sind Makros, deren Expansion zu neuen Typen, Funktionen oder Methoden führt. Im Vergleich zu klassischen Makros bleibt es jedoch nicht bei einer rein textuellen Ersetzung, da expandierte Templates vom Typechecker überprüft werden.

Bedingungen an ein Typargument sind implizit im Programmcode eines Templates verborgen. Da Templates keinen Typ besitzen, werden eventuelle Fehler erst nach der Expansion des Tem-

plates erkannt. Der Template-Code muß daher zur Übersetzungszeit als Quelltext vorliegen, was eine separate Übersetzung von Templates ausschließt[4].

Vergleichen wir abschließend mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$. In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ können polymorphe Funktionen durch Einsatz von `Typecase` auf ganz bestimmte Typen eingeschränkt werden. Im Gegensatz zu System $\mathbf{F}_{<:}$ ist es jedoch nicht möglich, allein anhand des Typs einer polymorphen Funktion zu entscheiden, welche Typargumente erlaubt sind. Wie wir am Beispiel der Überladung gesehen haben, wird eine Einschränkung vielmehr dadurch erreicht, daß man für unerwünschte Argumenttypen „unbrauchbare" Instanzen generiert. Sofern eine polymorphe Funktion allein durch `Typecase` eingeschränkt wird, ist eine separate Übersetzung möglich, sofern das Typprogramm zur Realisierung der Beschränkung vorliegt. Die Übersetzung von überladenen Funktionen, also polymorphen Funktionen, die in Abhängigkeit von der Struktur des Typarguments definiert sind, ist nicht möglich.

5.3 C++ und System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$

Wir wollen jetzt zeigen, daß es in C++ zu nahezu jedem System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$-Konstrukt ein Gegenstück gibt und sich C++ daher hervorragend als Modellsprache eignet, um Sprachintegration durch strukturelle Typanalyse einem Praxistest zu unterziehen.

5.3.1 Überladung - Zwittergestalt von C++ -Funktionen

Überladung ist in C++ und System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ähnlich realisiert. In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ müssen wir für eine überladene Funktion zwei Funktionen zur Verfügung stellen. Die erste beruht auf `Typecase` und berechnet den Typ einer Funktion in Abhängigkeit vom Typ des Arguments und schränkt die Funktion auf gültige Argumenttypen ein. Die zweite generiert, ausgehend vom Argumenttyp, mit Hilfe von `typecase` die passende Funktion.

Betrachtet man überladene C++ -Funktionen, so läßt sich eine ähnliche Zweiteilung beobachten, allerdings sind die einzelnen Funktionen syntaktisch nicht voneinander getrennt. Nehmen wir zum Beispiel die überladene Funktion **f**[5]:

```
int f(int x,int y) { return x + y; }
double f(double x) { return -x;    }
```

Der Ergebnistyp von `f` hängt von den Argumenttypen ab. Lassen wir Konvertierungen außer acht, können wir den Zusammenhang zwischen Argument- und Ergebnistyp mit einer System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Funktion beschreiben:

$$\begin{array}{lll} \mathtt{f}_{\mathbb{T}} := \Lambda\delta :: *.\mathtt{Match}\ \delta\ \mathtt{with} & \mathtt{int} \times \mathtt{int} & \Rightarrow \quad +_{\mathbb{T}}[\mathtt{int} \times \mathtt{int}] \\ & \mathtt{double} & \Rightarrow \quad -_{\mathbb{T}}[\mathtt{double}] \\ & \mathtt{else} & \mathtt{FAIL} \end{array}$$

[4]Prinzipiell sieht der C++ -Standard zwar vor, daß Template-Deklarationen als `export`, also in einer anderen Übersetzungseinheit befindlich, vereinbart werden können, jedoch ist zur Zeit kein Übersetzer verfügbar, der hierfür Unterstützung bietet.

[5]Wir konzentrieren uns hier auf solche C++ -Funktionen, die nur aus einem `return`-Statement bestehen, da sich diese direkt in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ darstellen lassen.

Diese Funktion nennen wir die **zu `f` gehörige Typfunktion**. Dabei sind $+_{\mathbb{T}}$ und $-_{\mathbb{T}}$ die zu den eingebauten Operatoren `operator+` und `operator-` gehörenden Typfunktionen und der Typ `FAIL` symbolisiert einen Typfehler.

Wird nun `f` durch Hinzunahme eines Funktionstemplates, etwa

```
template <typename B,typename C>
B f(B x1,B x2,C3 x3) { return x1 + x2 * x3; }
```

überladen, bedeutet dies die implizite Erweiterung von $\mathtt{f}_{\mathbb{T}}$ zu:

$$\begin{array}{lllll} \mathtt{f}_{\mathbb{T}} := \Lambda\delta :: *.\mathtt{Match}\ \delta\ \mathtt{with} & \mathtt{int} \times \mathtt{int} & \Rightarrow & +_{\mathbb{T}}[\mathtt{int} \times \mathtt{int}] \\ & \mathtt{double} & \Rightarrow & -_{\mathbb{T}}[\mathtt{double}] \\ & \alpha \times \alpha \times \beta & \Rightarrow & *_{\mathbb{T}}[+_{\mathbb{T}}[\alpha \times \alpha] \times \beta] \\ & \mathtt{else} & & \mathtt{FAIL} \end{array}$$

Neben der Typberechnung legt eine C++ -Funktionsdefinition eine Wertberechnung fest. Auch diese ist abhängig von den Argumenttypen. In System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ kann man `f` als Funktion darstellen, die einen Typ als Eingabe erwartet, und eine Funktion als Ergebnis zurückliefert. Diese Funktion nennen wir die **zu `f` gehörende Codeselektionsfunktion**:

$$\begin{array}{lllll} \mathtt{f}_{\mathbb{V}} := \Lambda\delta :: *.\mathtt{match}\ \delta\ \langle\Lambda\gamma :: *.\mathtt{f}_{\mathbb{T}}[\gamma]\rangle\ \mathtt{with} & & & & \\ & \mathtt{int} \times \mathtt{int} & \Rightarrow & +_{\mathbb{V}}[\mathtt{int} \times \mathtt{int}] & \\ & \mathtt{double} & \Rightarrow & -_{\mathbb{V}}[\mathtt{int} \times \mathtt{int}] & \\ & \alpha \times \alpha \times \beta & \Rightarrow & \lambda x : \alpha \times \alpha \times \beta. & \\ & & & *_{\mathbb{V}}[+_{\mathbb{T}}[\alpha \times \alpha] \times \beta](+_{\mathbb{V}}[\alpha \times \alpha]\ x.0\ x.1, x.2) & \end{array}$$

Die Funktionen $+_{\mathbb{V}}$, $*_{\mathbb{V}}$ und $-_{\mathbb{V}}$ sind die Codeselektionsoperationen für die in C++ eingebauten Operatoren `operator+`, `operator*` und `operator-`.

Man kann sich die Arbeit des C++ -Übersetzers nun so vorstellen, daß er für jeden Funktionsbezeichner eine Codeselektions- und eine Typfunktion generiert und sämtliche Funktionsaufrufe wie Überladungsapplikationen behandelt:

```
int a,b;
f(a,b)  ⇒   f_V [[a,b]]
```

Da in C++ , wie in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$, durch Typelimination übersetzt wird, muß die Typapplikation zur Übersetzungszeit ausgewertet werden.

5.3.2 Konstruktortypen

Mit Hilfe von Klassentemplates können Konstruktortypen simuliert werden. Nullstellige Konstruktoren entsprechen dabei normalen Klassendefinitionen während sich n-stellige Konstruktoren als Klassentemplates mit n-Typparametern darstellen lassen. Konstruktorwerte entsprechen Objekten, wobei die Werte der Konstruktorkomponenten als Datenelemente des Objekts dargestellt werden. Klassentemplates, die System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Konstruktoren repräsentieren, halten daher zu jedem Typparameter ein passendes Datenelement bereit.

Mit dieser Analogie lassen sich z.B. Polytyplisten in C++ durch die in Abbildung 5.6 gezeigten Datentypen darstellen.

```
struct N {};   // Leere Liste
template <typename H,typename T>
struct C {
  // C++ Konstruktor
  C(const H& _h,const T& _t) : h(_h),t(_t) {}
  H h;  // Kopfelement
  T t;  // Restliste
};
```

Abbildung 5.6: Datentypen zur Darstellung von Polytyplisten in C++ .

Anmerkung 5.1. Klassen können in C++ auch über das Schlüsselwort `struct` vereinbart werden. Im Vergleich zu Klassen, die mit `class` definiert wurden, sind hier alle Elemente der Klasse per Vorgabe als `public` vereinbart, was uns im weiteren Verlauf ein wenig Schreibarbeit spart. ◊

Instanzen des Klassentemplates `C` speichern das Kopfelement einer Polytypliste im Datenfeld `h` und die Restliste im Datenfeld `t`. Polytyplisten (also Objekte einer Instanz von `C`) können über einen Konstruktor erstellt werden, der Kopfelement und Restliste als Argumente erwartet.

Eine Polytypliste, die einen Wert vom Typ `int` und einen Wert vom Typ `bool` speichert hat in C++ den Typ

$$\texttt{C} < \texttt{int}, \texttt{C} < \texttt{bool}, \texttt{N} > >$$

Im Vergleich zu System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$werden beim Typnamen spitze Klammern, anstelle von Runden Klammern eingesetzt. Weitere Unterschiede ergeben sich bei der Beschreibung von Werten. In C++ ist `C(3,C(true,N))` kein gültiger Term, da `C` kein gültiger Datentyp, sondern ein Template ist, das zunächst instanziiert werden muß. Darüberhinaus ist auch `N` kein gültiger Wert, sondern ein Typ. Korrekt ist in C++ der folgende Term:

$$\texttt{C} < \texttt{int}, \texttt{C} < \texttt{bool}, \texttt{N} > > (\texttt{3}, \texttt{C} < \texttt{bool}, \texttt{N} > (\texttt{true}, \texttt{N}()))$$

Bevor ein Wert vom Typ `C<int,C<bool,N> >` erstellt werden kann, muß das Klassentemplate `C` instanziiert werden – dazu kommt die uns schon bekannte Notation mit spitzen Template-Klammern zum Einsatz. Das Erzeugen des Werts selbst läuft über den Aufruf des Konstruktors der jeweiligen Klassen[6].

Im direkten Vergleich mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$verhält sich das Template `C` wie die System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Funktion `cons`, die wir auf Seite 51 bereits eingeführt hatten:

$$\texttt{cons} = \Lambda\alpha :: *.\Lambda\beta :: *.\lambda x : \alpha.\lambda y : \beta.\texttt{C}(x, y)$$

Unter Einsatz von `cons` sieht die Konstruktion des Werts `C(3,C(true,N))` der C++ -Variante sehr ähnlich:

$$\texttt{cons}[\texttt{int}][\texttt{C}(\texttt{bool}, \texttt{N})]\ \texttt{3}\ \texttt{cons}[\texttt{bool}][\texttt{N}]\ \texttt{true}\ \texttt{N}$$

[6]Für die Klasse `N` erzeugt der C++ -Übersetzer automatisch einen default-Konstruktor.

Im wesentlichen wurden Template-Klammern durch Typapplikationsklammern ersetzt. Man kann aber auch die einfachere System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Schreibweise `C(3,C(true,N))` in C++ ermöglichen, indem man nachfolgendes Funktionstemplate bereitstellt:

```
template <typename H,typename T>
C<H,T> cons(const H& h,const T& t) {
  return C<H,T>(h,t);
}
```

Durch Aufruf von `cons` kann der gewünschte Konstruktorwert nun auch in C++ ohne explizite Typannotationen erzeugt werden:

$$\texttt{cons}(3, \texttt{cons}(\texttt{true}, \texttt{N}()))$$

Ein Verzicht auf Typannotationen ist möglich, da der C++ -Übersetzer die Typparameter im Zuge der Auflösung von Überladung und der Funktionstemplate-Expansion automatisch an die konkreten Argumenttypen bindet.

Das erzeugte Objekt ist ein **temporäres Objekt**, dessen Lebenszyklus mit der Auswertung des Ausdrucks endet, innerhalb dessen es erzeugt wurde. Will man ein mit `cons` erzeugtes Objekt persistent speichern (z.B. in einer Variablen), so kommt man nicht umhin, seinen Typ anzugeben:

```
Cons<int,Cons<double,NIL> > var = cons(2, cons(2.56, NIL() ) );
```

Das macht insbesondere lange Polytyplisten sehr unhandlich.

In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ stellt sich das Problem natürlich nicht, da es zum einen keinen globalen Speicher gibt, und in `let`-Ausdrücken der Typ einer `let`-Variablen durch den zugehörigen Ausdruck bestimmt wird:

$$\begin{array}{l}\texttt{let}\ l = \texttt{C}(4, \texttt{C}(\texttt{true}, \texttt{N})) \\ \quad\ \texttt{in}\ \cdots\end{array}$$

Die Typregel **T-Let** (siehe Seite 47) weist der Variablen l den Typ `C(int,C(bool,N))` zu.

Neben Listen werden wir im nächsten Abschnitt die Darstellung von natürlichen Zahlen durch geschachtelte Konstruktoren benötigen. Sei `S` ein einstelliger Konstruktor. Dann können wir die Null durch den Konstruktor `N` und Zahlen $n > 0$ durch n-fache Anwendung von `S` auf `N` darstellen. Dann steht zum Beispiel `S(N)` für die Eins, `S(S(N))` für die Zwei usw. `S` (für *successor* – Nachfolger) und `N` kann man natürlich wieder als Klasse bzw. Klassentemplate realisieren, so daß der Typ `N` der Null, der Typ `S<N>` der Eins, der Typ `S<S<N> >` der Zwei entspricht und so fort.

Im nächsten Abschnitt zeigen wir, wie man Λ-Abstraktion und Typapplikationen mit Hilfe von C++ -Templates simulieren kann, um so mit Zahlen in dieser Darstellung zu rechnen.

5.3.3 Typabhängige Typen in C++

Wie in System $\mathbf{F}_\omega$, gibt es in C++ das Konzept äquivalenter Typen. In einfachsten Fall können über `typedef` Definitionen Aliase für einen bestehenden Typ vereinbart werden. Zum Beispiel führt die Deklaration

```
typedef Stack<int> intStackType;
```

den neuen Typnamen `intStackType` als alternativen Namen für den Typ `Stack<int>` ein und es gilt:

$$\texttt{instStackType} \equiv \texttt{Stack} < \texttt{int} >$$

Im Zusammenhang mit Klassentemplates erwächst aus `typedef`-Deklarationen ein ausdrucksstarker Mechanismus, der den Typfunktionen in System $\mathbf{F}_\omega$ gleichkommt. Typaliasdefinitionen dürfen nämlich auch im Sichtbarkeitsbereich von Klassen und Klassentemplates eingesetzt werden:

```
template <class T>
struct PlusOne {
  typedef S<T>      RET;
};
```

Der Typalias `PlusOne` ist abhängig vom Typ-Parameter `T`.

Das Klassentemplate `PlusOne` kann man unter verschiedenen Blickwinkeln betrachten. Unter der gebräuchlichsten Sichtweise, der nahezu alle C++ -Bücher folgen, ist `PlusOne` ein generischer Datentyp. Man kann `PlusOne` aber auch als Funktion auffassen, die ein Typargument übernimmt und als Ergebnis einen Verbundtyp mit dem Feld `RET` liefert. Nimmt man diesen Standpunkt ein, so nennt man das Klassentemplate `PlusOne` eine **Template-Metafunktion**.

Diese Interpretation von `PlusOne` wird vielleicht etwas klarer, wenn man sich eine äquivalente Funktion in System $\mathbf{F}^{SA}_{\omega,1}$anschaut:

$$\texttt{PlusOne} := \Lambda\alpha :: *.\texttt{S}(\alpha)$$

`PlusOne` übernimmt einen Typ als Argument und umschließt dieses mit dem Konstruktor `S`. Sofern wir eine gültige Zahl (in der Darstellung mit `N` und `S`) an `PlusOne` übergeben, macht die Funktion genau das, was ihr Name verheißt: Das Argument wird um Eins erhöht.

Der System $\mathbf{F}^{SA}_{\omega,1}$-Term `PlusOne[S(N)]` entspricht in C++ dem Term

```
PlusOne<S<N> >::RET
```

Die äußeren spitzen Klammern in C++ entsprechen den eckigen Klammern der Typapplikation in System $\mathbf{F}^{SA}_{\omega,1}$. Ansonsten sind die syntaktischen Unterschiede eher marginal. Lediglich der Zugriff auf den Ergebniswert verläuft verschiedenartig. In System $\mathbf{F}^{SA}_{\omega,1}$ wird er unmittelbar von der Funktion zurückgeliefert, in C++ muß man mit dem Scope-Operator `::` explizit auf den Feldwert von `RET` zugreifen.

Es lassen sich auch komplexere Typfunktionen realisieren. Betrachten wir dazu eine System $\mathbf{F}^{SA}_{\omega,3}$-Funktion zur Addition zweier Zahlen:

$$\begin{array}{lll}
\texttt{Add} := & \texttt{fix}\ \Lambda\texttt{addRec} :: * \Rightarrow *.\Lambda\alpha :: *. & \\
& \texttt{Match}\ \alpha\ \texttt{with} & \texttt{N} \times \alpha \quad \Rightarrow \quad \alpha \\
& & \texttt{S}(\alpha) \times \beta \quad \Rightarrow \quad \texttt{S}(\texttt{addRec}[\alpha \times \beta]) \\
& & \texttt{else} \quad \Rightarrow \quad \texttt{FAIL}
\end{array}$$

```
Add := Λδ :: ∗.Typecase δ of
                default :  Λα :: ∗.α
                ×       :  Λα :: ∗.Λα′ :: ∗,
                           Λβ :: ∗.Λβ′ :: ∗.
                           let doAdd = Typecase α of
                                          N :  Λγ :: ∗.γ
                                          S :  Λα :: ∗.Λα′ :: ∗ ⇒ ∗.
                                               Λγ :: ∗.S(α′[γ])
                              in doAdd[β]
```

Abbildung 5.7: Skizze einer auf `Typecase` basierenden Funktion zur Addition. Der Übersicht halber gehen wir in diesem Beispiel davon aus, daß `let`-Ausrücke auch für Typen und Typfunktionen möglich sind.

Der im `else`-Fall zurückgegebene Wert `FAIL` möge wieder einen Typfehler auslösen. Die Funktion `Add` ist rekursiv definiert und nutzt *pattern matching* zur Analyse des Funktionsarguments. Wir setzen hier bewußt den Fixpunktkombinator ein, um die Rekursion explizit sichtbar zu machen. `Add` könnte natürlich auch in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ unter alleiniger Verwendung von `Typecase` implementiert werden (siehe Abbildung 5.7).

Das *pattern matching* für Typen kann in C++ über Template-Spezialisierung erreicht werden. Zur Umsetzung von rekursiven Funktionen bedarf es im Vergleich zu System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ jedoch keines Fixpunktkombinators, da eine Funktion (ein Klassentemplate) direkt über seinen Namen angesprochen werden kann:

```
template <class X,class Y>     /* 0 */
struct Add;

template <class X>             /* 1 */
struct Add<N, X>    { typedef N RET; };

template <class X,class Y>     /* 2 */
struct Add<S<X>, Y> { typedef S< typename Add<X,Y>::RET > RET; };
```

Das Muttertemplate entspricht dem `else`-Fall der System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$-Funktion. Sollte `Add` im ersten Argument mit einem Typ aufgerufen werden, der nicht ausschließlich aus `S` und `N` aufgebaut ist, so müßte das Muttertemplate instanziiert werden. Da dieses aber nur deklariert und nicht definiert ist, kommt es zu einem Typfehler[7].

Die erste Template-Spezialisierung entspricht dem ersten Behandlungsfall des `Match`-Terms im System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$-Term. Wird Null zu einem Wert `X` addiert, so ist das Ergebnis `X`. Die zweite Spezialisierung deckt den Fall ab, daß eine Zahl größer Null zu einer Zahl `Y` addiert werden soll. Im Vergleich zu System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ können wir beim rekursiven Aufruf die Funktion `Add` direkt ansprechen, um an das Ergebnis der Funktionsapplikation zu gelangen muß jedoch wieder der *scope*-Operator eingesetzt werden, um den Feldwert von `RET` zu erfragen.

Das vorangestellte Schlüsselwort `typename` gilt dem C++ -Übersetzer als Hinweis, daß es

[7] Man könnte durch Hinzunahme von weiteren Mustern auch das zweite Argument entsprechend einschränken, um so ein gewisses Maß an „Typsicherheit" zu erreichen

sich bei `RET` um einen Typ handelt. Obwohl vom Standard vorgeschrieben, akzeptieren viele Übersetzer, wenn man es ausläßt. Im Sinne einer kürzeren und übersichtlicheren Schreibweise werden auch wir dem manchmal folgen.

Der C++ -Template-Mechanismus übernimmt z.T. die Aufgaben des partiellen Auswerters $\mathcal{T}$, den wir als integralen Bestandteil von System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ verstehen: Template-Instanziierungen entsprechen der β-Reduktion von Typapplikationen in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ und werden zur Übersetzungszeit vollzogen.

Zum Beispiel „reduziert" der C++ -Übersetzer den Ausdruck `Add<S<S<N> >,S<N> >::RET` wie folgt zu `S<S<S<N> > >`:

$$\begin{aligned} & \texttt{Add} < \texttt{S} < \texttt{S} < \texttt{N} > >, \texttt{S} < \texttt{N} > >:: \texttt{RET} \\ \xrightarrow{2} \quad & \texttt{S} < \texttt{Add} < \texttt{S} < \texttt{N} >, \texttt{S} < \texttt{N} > >:: \texttt{RET} > \\ \xrightarrow{2} \quad & \texttt{S} < \texttt{S} < \texttt{Add} < \texttt{N}, \texttt{S} < \texttt{N} >:: \texttt{RET} >> \\ \xrightarrow{1} \quad & \texttt{S} < \texttt{S} < \texttt{S} < \texttt{N} > > > \end{aligned}$$

Mit $\xrightarrow{2}$ ist gemeint, daß der Übersetzer seine Arbeit unter Expansion der zweiten Template-Spezialisierung fortsetzt. Ohne daß man auf den Typalias `RET` zugreift, passiert allerdings gar nichts. Dann greift das Prinzip der verzögerten Template-Expansion und da der Wert von `RET` nicht angesprochen wird, wird auch keine Normalform zu diesem Typalias bestimmt.

Die Strategie des *pattern matching* ist in beiden Sprachen unterschiedlich. In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ werden Muster nacheinander geprüft und es wird immer das erste Muster ausgewählt, das zum untersuchten Term paßt. In C++ hingegen wird das Muster gewählt, welches hinsichtlich der Matchordnung am speziellsten ist – die Reihenfolge der Muster im Quelltext ist für das Ergebnis bedeutungslos.

Wir wollen den Unterschied an einem Beispiel verdeutlichen (siehe Abbildung 5.8).

In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ wird der Term `Foo[ S(S(N)) ]` zu `S(N)` reduziert, da der zweite Behandlungsfall des `Match`-Terms greift und α an `S(N)` gebunden wird.

In C++ erhält man ein anderes Ergebnis. Hier führt die Reduktion von `Foo<Succ<Succ<Null> > >::RET` zum Typ `Null`. Zunächst passen die Muster der zweiten und dritten Spezialisierung mit

$$subst = \{\texttt{X} \mapsto \texttt{S} < \texttt{Null} >\}$$

bzw.

$$subst = \{\texttt{X} \mapsto \texttt{Null}\}$$

Die dritte Spezialisierung ist jedoch spezieller als die erste, denn bezüglich der Matchordnung (siehe Seite 27) gilt:

$$\texttt{S} < \texttt{X} > \trianglelefteq \texttt{S} < \texttt{S} < \texttt{X} > >$$

Der gegenüber System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ flexiblere Mechanismus hat insbesondere den Vorteil, daß Funktionen leichter um neue Fälle erweitert werden können. Insbesondere müssen Template-Spezialisierungen nicht in derselben Quelltextdatei definiert sein, wie das Muttertemplate. Auch

System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

$$\begin{array}{lll} \texttt{Foo} := \Lambda\alpha :: *.\texttt{Match}\ \alpha\ \texttt{with} & \texttt{N} & \Rightarrow\ N \\ & \texttt{S}(\alpha) & \Rightarrow\ \alpha \\ & \texttt{S}(\texttt{S}(\alpha)) & \Rightarrow\ \alpha \end{array}$$

C++

```
template <class T> struct Foo; /* 0 */

template <>
struct Foo<N> {             /* 1 */
  typedef N RET;
};

template <class X>            /* 2 */
struct Foo<S<X> > {
  typedef X RET;
};

template <class X>           /* 3 */
struct Foo<S<S<X> > >  {
  typedef X RET;
};
```

Abbildung 5.8: Funktion `Foo` in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$(oben) und als C++ -Typfunktion (unten).

wenn keine separate Übersetzung von Templates möglich ist, bietet sich doch die Möglichkeit, Bibliotheken z.B. über Header-Dateien einzubinden und extern zu erweitern ([33]).

Als abschließendes Beispiel stellen wir die Berechnung der Länge einer Liste in Haskell, System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ und als Template-Metafunktion `length` gegenüber (siehe Abbildung 5.9).

5.3.4 Typabhängige Terme in C++

Es bringt uns natürlich recht wenig, wenn wir die Ergebnisse von Typberechnungen nicht zur Laufzeit-Ebene transportieren können. Was sollen wir zum Beispiel mit dem Umstand anfangen, daß `Add<S<N>, N>::RET` den Wert `S<N>` liefert? Wir können diesen Wert weder auf dem Bildschirm ausgeben, noch in eine Datei schreiben, geschweige denn, ihn an eine Laufzeitfunktion übergeben.

In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ können wir das Problem recht einfach lösen, da Wertfunktionen auch von Typen abhängen können. Eine durch die Konstruktoren `N` und `S` repräsentierte Zahl läßt sich mit `match` recht einfach in einen `int`-Wert konvertieren:

$$\begin{array}{lll} \texttt{eval} := \Lambda\alpha :: *.\texttt{match}\ \alpha\ \texttt{with} & \texttt{N} & \Rightarrow\ 0 \\ & \texttt{S}(\alpha) & \Rightarrow\ 1 + \texttt{eval}[\alpha] \end{array}$$

Der zu einer Zahl in Typdarstellung passende Wert ergibt sich gerade aus der Schachtelungstiefe des Konstruktorterms (Anmerkung: eine mit `typecase` realisierte Variante dieser Funktion hatten wir bereits auf Seite 59 vorgestellt).

C++

```
class N {};

template <class H,class T>
struct C {};


template <class L> struct Length;


template <>
struct Length<N> {
  typedef N RET;
};


template <class H,class T>
struct Length<C<H,T> > {
  typedef typename
  Add<EINS,
      typename Length<T>::RET
     >::RET RET;
};
```

Haskell

```
data List a =  N
             | C a (List a)


length :: List a -> int


length N  = 0


length (C h t) = 1+(length t)
```

System $F^{SA}_{\omega,1}$

$$\texttt{length} := \Lambda\delta :: *.\texttt{Match}\ \delta\ \texttt{with} \quad \begin{array}{lcl} \texttt{N} & \Rightarrow & \texttt{N} \\ \texttt{C}(\alpha,\beta) & \Rightarrow & \texttt{Add}[\ \texttt{S(N)} \times \texttt{length}[\beta]\ ] \end{array}$$

Abbildung 5.9: Vergleich: Berechnung der Länge einer Polytypliste in C++ (oben links) und System $\mathbf{F}^{SA}_{\omega,1}$ (unten), sowie der Berechnung der Länge einer Liste in Haskell (oben rechts).

In C++ können wir denselben Effekt erzielen, wenn wir Typfunktionen (Klassen und Klassentemplates) mit statischen Methoden ausstatten. Das sind Methoden, die ohne ein Objekt aufgerufen werden können – eine Klasse übernimmt hier mehr oder weniger die Rolle eines Namensraums.

```
template <class T>
struct Eval;

struct Eval<N> {
  static int exec() { return 0; }
};
```

Als Metafunktion betrachtet ist `Eval` eine einstellige Funktion, die im Feld `exec` einen Zeiger auf eine parameterlose C++ -Funktion zurückgibt. Abfragen des Feldwertes von `exec` (z.B. durch `Eval<N>::exec`) hat zur Folge, daß die Methode `exec` expandiert wird, d.h. es wird Programmcode generiert und ein Funktionszeiger auf diesen Code als Feldwert von `exec` zurückgegeben. Insofern entspricht der Vorgang der Term-Typ-Applikation in System **F**.

Wird auf den zurückgelieferten Funktionszeiger der Klammeroperator angewendet, so generiert der C++ -Übersetzer Programmcode, der die soeben kreierte Funktion aufruft:

```
(Eval<N>::exec)();
```

Sofern der Übersetzer das *inlining* von Funktionen unterstützt, kann er alternativ auch den Funktionsrumpf von `Eval<Null>::exec` an der Stelle des Funktionsaufrufs expandieren. Als Effekt würde das Codefragment `(Eval<Null>::exec)()` unmittelbar durch die Konstante `0` ersetzt. Template-Expansion und *inlining* entsprechen dann der partiellen Auswertung mit $\mathcal{T}$ in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

Für C++ -Programmierer erscheint diese Art der Betrachtung im ersten Moment wahrscheinlich ungewöhnlich, da bereits die Schreibweise

```
(Eval<N>::exec)()
```

von der gebräuchlichen Notation

```
Eval<N>::exec()
```

abweicht.

Läßt man sich aber auf den Gedanken ein, daß `Eval<Null>::exec` zunächst eine Funktion *generiert* und einen Zeiger auf diese als Ergebnis zurückgibt, erscheint die erstere Schreibweise angemessener.

Mit Hilfe von Template-Spezialisierung können wir jetzt auch `typecase`-Terme auf C++ abbilden. Um Zahlen größer Null auf dem Bildschirm ausgeben zu können, spezialisieren wir das Klassentemplate `Eval`; d.h. wir ergänzen die Metafunktion `Eval` um einen Behandlungsfall für „Zahlen", die mit dem Konstruktor `S` erzeugt wurden:

```
template <class T>
struct Eval<S<T> > {
  static inline int exec() {
    return 1 + (Eval<T>::exec)();
  }
};
```

Wie zuvor, gibt die Funktion `Eval` im Feldnamen `exec` einen Zeiger auf eine parameterlose C++ -Funktion zurück. Das Funktionsergebnis ist jedoch abhängig vom Typ, der an die Variable `T` gebunden wurde. Daher muß zunächst Programmcode zur Berechnung des Wertes von `T` generiert werden – der Ausdruck `Eval<T>::exec` liefert die Adresse einer entsprechenden C++ -Funktion.

Betrachten wir an einem Beispiel, wie der C++ -Übersetzer den Ausdruck

```
(Eval<S<S<N> > >::exec)()
```

abarbeitet. Im ersten Schritt wird Programmcode für die Funktion `exec` generiert. Der Übersetzer instanziiert hierzu das `Eval` Klassentemplate und dessen `exec` Methode. Dabei entsteht eine neue Klasse:

```
struct Eval__1 { // Eval<S<S<N> > >
  static int exec() {
    return 1 + (Eval<S<N> >::exec)();
  }
};
```

Um die Klasse `Eval__1` auf Typkonsistenz hin untersuchen zu können, muß der Rückgabetyp der vom Metaprogramm `(Eval<S<N> >::exec)` generierten Funktion vorliegen. Außerdem hängt der Programmcode für die `exec`-Methode vom Programmcode der `exec`-Methode der Klasse `Eval<S<N> >` ab – der Programmcode für das Sprungziel fehlt jedoch noch. Der Übersetzer ist daher gezwungen, im nächsten Schritt den Template-Meta-Term `Eval<S<N> >::exec` abzuarbeiten, was bedeutet, daß das Klassentemplate `Eval` und seine statische Methode `exec` erneut expandiert werden müssen:

```
struct Eval__2 { // Eval<S<N> >
  static int exec() {
    return 1 + (Eval<N>::exec)();
  }
};
```

Aus denselben Gründen wie zuvor, muß zunächst der Aufruf der Metafunktion `Eval<N>::exec` abgearbeitet werden — abermals wird das `Eval` Klassentemplate expandiert; diesmal jedoch aus dem Programmcode der ersten Spezialisierung:

```
struct Eval__3 { // Eval<N>
  static int exec() {
    return 0;
  }
};
```

Weder die Prüfung auf Typkonsistenz, noch das Generieren von Programmcode für die Methode `exec` bedarf in diesem Fall einer weiteren Template-Expansion.

Jetzt sind die Sprungziele in den Rümpfen der `exec`-Methoden der Klassen `Eval__1` und `Eval__2` bekannt und deren Definition kann entsprechend angepaßt werden:

```
struct Eval__1 { // Eval<S<S<N> > >
  static int exec() {
    return 1 + (Eval__2::exec)();
  }
};
```

Der entstandene Programmcode ist frei von Typapplikationen und kann durch Typelimination übersetzt werden.

Abbildung 5.10 veranschaulicht den Vorgang nochmal grafisch, wobei wir im Sinne einer einfacheren Darstellung die folgenden Typaliasdefinitionen vorausgesetzt haben[8].

[8] Im weiteren Verlauf gehen wir davon aus, daß entsprechende Definitionen für `DREI`, `VIER`, `FUENF` usw. vereinbart wurden.

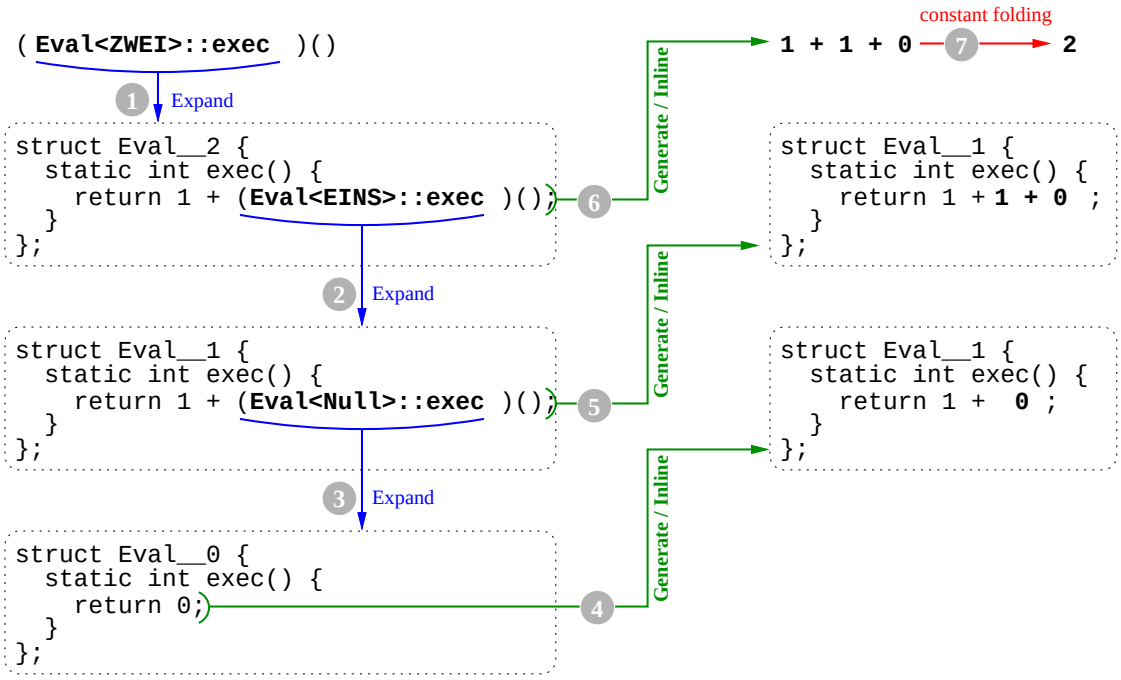


Abbildung 5.10: Generierung von Programmcode für `(ZWEI::exec)()`

```
typedef S<N>            EINS;
typedef S<S<N> >         ZWEI;
```

Die Auswertung von `(Eval<ZWEI>::exec)()` verläuft in vier Phasen. In der ersten Phase werden die jeweiligen Klassentemplates expandiert (Schritte 1 bis 3). Die zweite und dritte Phase verlaufen verzahnt. Zunächst wird die Adresse der generierten Funktion (bzw. ihr Name) an die Aufrufposition geschrieben. Zum Beispiel wird in Schritt 5 der Ausdruck `Eval<Null>::exec` durch den Ausdruck `Eval__0::exec` ersetzt. Diesen kann der Übersetzer durch Anwendung von *inlining* direkt zu `return 1+0` vereinfachen. In der vierten Phase wird der gerade generierte Code optimiert. In unserem Beispiel kann er zum Beispiel durch *constant folding* zu einer einfachen Konstanten reduziert werden (Schritt 7).

Jetzt, wo wir in der Lage sind, Ergebnisse aus Typberechnungen für Laufzeitfunktionen zugänglich zu machen, können wir das Ergebnis einer Meta-Addition auf die Standard-Ausgabeeinheit schreiben:

```
cout << (Eval<Add<DREI, ZWEI>::RET>::exec) () << endl;
```

Der Übersetzer muß hier zwei ineinandergeschachtelte Meta-Terme abarbeiten. Zuerst wird das Ergebnis von `Add<DREI,ZWEI>::RET` berechnet:

$$
\begin{array}{ll}
 & \texttt{Add} < \texttt{ZWEI}, \texttt{DREI} >:: \texttt{RET} \\
\xrightarrow{2} & \texttt{S} < \texttt{Add} < \texttt{EINS}, \texttt{DREI} >:: \texttt{RET} > \\
\xrightarrow{2} & \texttt{S< S<} \quad \texttt{Add} < \texttt{NULL}, \texttt{DREI} >:: \texttt{RET} \quad \texttt{> >} \\
\xrightarrow{1} & \texttt{S< S< DREI > > >} \\
= & \texttt{FUENF}
\end{array}
$$

Damit wird der Ausdruck

```
(Eval<Add<DREI,ZWEI>::RET>:exec)()
```

zu

```
(Eval<FUENF>::exec)()
```

reduziert und der Übersetzer muß den Feldwert `exec` bestimmen – das verläuft analog zu Abbildung 5.10 und wir erhalten im Ergebnis:

```
cout << 5 <<endl;
```

Der Effekt entspricht einer einfachen, benutzerdefinierten Programmtransformation.

Typfunktionen als *partial evaluator*

Wir wollen nun ein Beispiel betrachten, bei dem die Strukturinformation allein nicht ausreicht, um ein Problem zu lösen.

Unsere Aufgabe sei, einen bestimmten Wert in einer Polytypliste zu suchen. Im Vergleich zur Suche in einer monomorphen Liste müssen wir dabei aufpassen, daß wir zwei Elemente nur dann miteinander vergleichen, wenn sie von identischem Typ sind.

Abbildung 5.11 zeigt die Implementierung einer entsprechenden Funktion in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$. Die Funktion `elem` bekommt das zu suchende Element in m und die Polytypliste in l übergeben. Zunächst wird die Liste einer `case`-Analyse unterzogen, so daß wir Zugang zu Kopfelement und Restliste erhalten, War die Liste leer, so konnte m nicht gefunden werden und es wird `false` als Ergebnis zurückgeliefert. Stimmen Kopfelementtyp und der Typ von m überein, kann verglichen werden und es muß ggf. mit der Suche in der Restliste fortgefahren werden. Paßt der Typ des Kopfelements nicht zu m, wird nur in der Restliste gesucht – das Kopfelement bleibt unberührt.

Für eine Implementierung in C++ starten wir mit einem Metaprogramm, das die Äquivalenz zweier Typen testet. Boolesche Werte stellen wir auf der Meta-Ebene durch die Typen `TRUE` und `FALSE` dar:

```
class TRUE {};
class FALSE {};

template <class X,class Y>
struct Equiv { typedef FALSE RET; };

template <class X>
struct Equiv<X,X> { typedef TRUE RET; };
```

$$
\begin{array}{lll}
\texttt{elem} := \Lambda\delta :: *.\lambda m : \delta. & & \\
\quad \Lambda\eta :: *.\lambda l : \eta. & & \\
\quad \texttt{case}\ l : \eta\ \texttt{of} & & \\
\quad \langle \texttt{N}\ ;\ x\rangle & \Rightarrow & \texttt{false} \\
\quad \langle \texttt{C}(\alpha,\beta)\ ;\ \texttt{C}(x,y)\rangle & \Rightarrow & \texttt{match}\ \alpha\ \texttt{with} \\
\quad\quad & & \texttt{default} \Rightarrow \texttt{elem}\,[\![m]\!]\,[\![y]\!] \\
\quad\quad & & \delta \Rightarrow \texttt{eq}\,[\![x,m]\!]\ ||\ \texttt{elem}\,[\![m]\!]\,[\![y]\!]
\end{array}
$$

Abbildung 5.11: Suche nach einem Element n vom Typ δ in einer Polytypliste l vom Typ η in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

Die innere `match`-Anweisung des System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Terms müssen wir auf eine separate Template-Metafunktion abbilden, da wir keine explizite `match`-Anweisung zur Verfügung haben und diese durch Template-Spezialisierungen simulieren müssen.

```
template <class EQ,class M,class H>
struct cmpHead;

template <class M,class H>
struct cmpHead<FALSE,M,H> {
  static bool exec(const M& m,const H& h) {
    return false;
  }
};

template <class M,class H>
struct cmpHead<TRUE,M,H> {
  static bool exec(const M& m,const H& h) {
    return m==h;
  }
};
```

Die Metafunktion `cmpHead` generiert im Feld `exec` den für den Vergleich notwendigen Programmcode. Das erste Argument an `cmpHead` signalisiert, ob Kopfelement und gesuchtes Element übereinstimmen. Wenn ja (zweite Spezialisierung), so wird der Vergleich vollzogen, ansonsten wird `false` zurückgegeben.

Abbildung 5.12 zeigt schließlich die Implementierung der Suchfunktion in C++ . Ein Äquivalent zu `case`-Termen gibt es in C++ leider nicht. Allerdings ist die Projektion in C++ komfortabler, da Konstruktorkomponenten nicht über Indizes, sondern über Feldnamen angesprochen werden können.

Betrachten wir nun folgendes Programmfragment:

```
template <class M,class L>
class Elem;

template <class M>
class Elem<M,N> {
  static
  bool exec(const M&,const N&) { return false; }
};

template <class M,class H,class T>
class Elem<M,C<H,T> > {
  static
  bool exec(const M& m,const C<H,T>& l) {
    return cmpHead< Equiv<M,H>::RET >::exec(m,l.h) || Elem<M,T>::exec(m,l.t);
  }
};

template <class M,class H,class T>
bool elem(const M& m,const C<H,T>& l) {
  return  Elem<M,C<H,T> >::exec( m, l);
}
```

Abbildung 5.12: Suche nach einem Element in einer Polytypliste in C++ .

```
int a;

cin >> a;

if ( elem(a, cons(''Hello'', cons(1, cons(2.56, NIL() ) ) ) ) )
  cout << ''found!'' << endl;
```

Der Aufruf der `elem` Funktion wird während der Übersetzungszeit reduziert. In Verbindung mit *inlining* entsteht der Term

```
false || a==1 || false
```

Weitere Optmierungen (z.B. *constant folding*) führen schließlich zu `a==1`.

Was ist passiert? Wir haben sämtliche zur Übersetzungszeit verfügbare Information ausgenutzt, um spezialisierten Programmcode zu erzeugen. Mit anderen Worten haben wir einen problemangepaßten partiellen Auswerter auf der Ebene des Typsystems realisiert.

Sonderstellung von Aufzählungstypen

Aufzählungstypen (*enum*-Typen) bilden eine Besonderheit in C++ , da die Elemente eines Aufzählungstyps in `int`-Werte konvertiert werden können und man jedes Element explizit mit einen `int`-Wert initialisieren kann. Daher lassen sich typabhängige integer-Konstanten auch ohne den Umweg der Codegenerierung berechnen:

```
template <typename T>
struct Eval;

template <>
struct Eval<N> {
  enum { Ret = 0 };
};

template <typename T>
struct Eval<S<T> > {
  enum { Ret = 1 + Eval<T>::Ret };
};
```

Der Ausdruck `Eval<S<S<N> > >::Ret` kann dann wie folgt direkt zum Wert Zwei reduziert werden:

$$
\begin{aligned}
& \texttt{Eval<S<S<N>>>::Ret} \\
\xrightarrow{2}\ & 1 + \texttt{Eval<S<N>>::Ret} \\
\xrightarrow{2}\ & 1 + 1 + \texttt{Eval<N>::Ret} \\
\xrightarrow{1}\ & 1 + 1 + 0 \\
\longrightarrow\ & 2
\end{aligned}
$$

Der letzte Reduktionsschritt ist keine Konstantenfaltung im klassischen Sinne, da sie vom Typsystem gewissermaßen vorgeschrieben ist.

Da Templates auch von integralen Werten abhängig sein dürfen, kann man in C++ sogar vollständig auf die Simulation von integer Zahlen mit `S`, `P` und `N` verzichten. Als Beispiel zeigen wir eine Metafunktion, mit der die Fakultät einer Zahl `T` zur Übersetzungszeit berechnet werden kann:

```
template <int T>
struct Fac;

template <>      struct Fac<0> { enum { Ret = 1 }; };
template <int T> struct Fac    { enum { Ret = 1 + Fac<T-1>::Ret }; };
```

5.3.5 Zusammenfassende Gegenüberstellung

Tabelle 5.13 stellt die syntaktischen und semantischen Eigenheiten von System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ und C++ gegenüber und zeigt Unterschiede bei der partiellen Auswertung.

Die meisten Punkte wurden im vorangegangenen Text bereits erklärt. An dieser Stelle wollen wir die Implementierung der einfachen Selektion für Typen in C++ nachtragen, zeigen wie man mit Klassentemplates Funktionen mit verzögerter Auswertung realisiert und noch ein paar ergänzende Worte zur partiellen Auswertung in C++ verlieren.

Einfache Selektion für Typen in C++

Die Implementierung der einfachen Selektion für Typen läßt sich sofort aus der Implementierung der im vorangegangen Abschnitt gezeigten Metafunktion `cmpHead` ableiten. Boolesche Werte werden dabei wieder als `TRUE` und `FALSE` dargestellt:

```
template <typename C,typename T,typename E>
struct If;

template <typename T,typename E>
struct If<TRUE,T,E> {
  typedef T RET;
};

template <typename T,typename E>
struct If<FALSE,T,E> {
  typedef E RET;
};
```

Wie in System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ ist die Typfunktion `If` dreistellig. Das Muttertemplate wurde lediglich deklariert, so daß das Instanziieren von `If` nur mit `TRUE` und `FALSE` als erstem Argument möglich ist. Das zweite Argument beschreibt den Rückgabetyp für den then-Fall (erstes Argument ist äquivalent zum Typ `TRUE`), das dritte Argument den else-Fall (erstes Argument ist äquivalent zum Typ `FALSE`).

Verzögerte Auswertung in C++

Grundsätzlich werden Typen in C++ zu Normalformen reduziert, bevor ein Template instanziiert wird. Da die eigentliche Funktionsapplikation aber zweistufig verläuft (Instanziieren des Templates, Abfrage eines Feldwert mit Hilfe des *scope*-Operators `::`), kann man die verzögerte Template-Expansion ausnutzen, um Typfunktionen mit verzögerter Auswertung zu realisieren.

Um die Idee zu verdeutlichen, betrachten wir zunächst die fehlerhafte Implementierung einer Funktion zur Berechnung der positiven Differenz $\ddot{-}$ zweier ganzer Zahlen ($x \ddot{-} := \max(x-y, 0)$) – siehe Abbildung 5.14.

Mit Hilfe des Klassentemplates `P` erweitern wir unsere Zahlendarstellung auf der Typebene um negative Zahlen. `P<N>` entspricht der -1, `P<P<N> >` der -2 und so weiter. Die Metafunktion `Dec` dekrementiert eine ganze Zahl.

Die positive Differenz wird von der Metafunktion `PDiff` berechnet. Soll eine Null von der ersten Zahl abgezogen werden, so ist das Ergebnis gerade die erste Zahl. War die erste Zahl die Null, so ist das Ergebnis Null, ansonsten wird `PDiff` mit jeweils dekrementierten Argumenten rekursiv aufgerufen.

Auf den ersten Blick sieht alles gut aus. Da Argumente an Template-Metafunktionen aber strikt ausgewertet werden, und wir `If` als Metafunktion realisiert haben, terminiert die Berechnung nicht!

Es gibt zwei Möglichkeiten diesen Fehler zu beheben. Entweder codiert man den Vergleich durch Template-Spezialisierungen, oder stellt eine Variante der Funktion `If` zur Verfügung,

Syntax		
Typfunktion	Klassen und Klassentemplates	$\Lambda\alpha :: *.T$ (Anonyme Funktion)
Typapplikation	`F<T>::RET`	$T_1[T_2]$
Rekursive Funktionen	Ja, Funktionen mit Namen	Ja, Fixpunktkombinator
Induktive Definition von Typen über Typnamen	Klassentemplates, Template-Spezialisierung, `typedef`	`Typecase` und `Match`
Induktive Definition von Funktionen über Typnamen	Klassentemplates, Template-Spezialisierung statische Methoden	`typecase` und `match`
Typäquivalenz	`Equiv<T1,T2>::RET`	$T_1 \equiv T_2$
Typselektion	`If<T1,T2,T3>::RET`	`If` T_C `then` T_T `else` T_E
`case`-Ausdrücke	Nein	Ja
`let`-Ausdrücke	Nein	Ja
Semantik		
Auswertungsstrategie	strikt (durch explizite Übergabe von *closures* auch nicht-strikt)	strikt
pattern matching	*best match*	*first match*
Partielle Auswertung		
Typapplikation, rekursive Typfunktionen Typselektion, Typäquivalenz	Ja	Ja
Projektionen	Compilerabhängig (*lightweight object optimization*)	Ja
`Typecase`	Ja	Ja
`typecase`	*pattern matching* evtl. *inlining*	*pattern matching* ein-Schritt β-Reduktion
Überladungsapplikation	Auflösen von Überladung evtl. *inlinig*	Typapplikation und ein-Schritt β-Reduktion

Abbildung 5.13: Gegenüberstellung von System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ und C++ .

```
struct N {};
template <typename T> struct S {};
template <typename T> struct P {};

template <typename T> struct Dec;
template <> struct
Dec<N>       { typedef P<N> RET; };
template <typename T> struct
Dec<P<T> >   { typedef P<P<T> > RET; };
template <typename T> struct
Dec<S<T> >   { typedef T RET; };

template <typename N1,typename N2>
struct PDiff {
  typedef typename
  If< typename Equiv<N2,N>::RET,
      N1,
      typename If<typename Equiv<N1,N>::RET,
                  N,
                  typename
                  PDiff< typename Dec<N1>::RET,
                         typename Dec<N2>::RET
                        >::RET
                 >::RET
    >::RET RET;
};
// Folgende Typberechnung terminiert nicht!
//typedef PDiff<S<N>,N>::RET RET;
```

Abbildung 5.14: Fehlerhaftes Metaprogramm zur Berechnung der positiven Differenz von ganzen Zahlen.

die auf **Funktionsabschlüssen** operiert. Unter einem Funktionsabschluß (engl. *closure*) versteht man ein Funktionssymbol zusammen mit einer Umgebung, welche den Variablen der Funktionsdefinition konkrete Werte zuweist.

Einen Funktionsabschluß können wir Dank der verzögerten Template-Expansion ganz einfach dadurch gewinnen, daß wir auf die Anwendung des *scope*-Operators nach der Funktionsapplikation verzichten.

Die Funktion `LazyIf` erwartet, daß ihr im zweiten und dritten Argument solche *closures* übergeben werden. Bevor der then- oder else-Zweig an den Aufrufer zurückgegeben werden, wird die Auswertung der *closure* durch Anwenden des *scope*-Operators angestoßen.

Falls ein Wert nicht über ein Feld `RET` verfügt, was z.B. auf alle bei der Zahlendarstellung involvierten Klassen zutrifft, müssen diese in die Klasse `Wrap` eingepackt werden, bevor sie an `LazyIf` übergeben werden (siehe Abbildung 5.15).

```
template <typename C,typename T,typename E>
struct LazyIf;
template <typename T,typename E>
struct LazyIf<TRUE,E,T>  {  typedef typename T::RET RET; };
template <typename T,typename E>
struct LazyIf<FALSE,E,T> {  typedef typename E::RET RET; };

template <typename T>
struct Wrap {
  typedef T RET;
};

template <typename N1,typename N2>
struct PDiff {
  typedef typename
  If< typename Equiv<N2,N>::RET,
      N1,
      typename LazyIf<typename Equiv<N1,N>::RET,
                      Wrap<N>,
                      PDiff< typename Dec<N1>::RET,
                             typename Dec<N2>::RET
                           >
                     >::RET
    >::RET RET;
};

typedef PDiff<S<N> ,N >::RET RET;
```

Abbildung 5.15: Korrekte Berechnung der positiven Differenz mit `LazyIf`.

Partielle Auswertung in C++ und System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$

In System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ ist garantiert, daß Typapplikationen, `Typecase`- und `typecase`-Terme, sowie Projektionen zur Übersetzungszeit partiell bzw. vollständig ausgewertet werden.

Um den Mehraufwand der in `typecase` inhärenten rekursiven Aufrufe auszugleichen, werden diese durch ein-Schritt-β-Reduktionen expandiert. Eine ein-Schritt-β-Reduktion wird ebenfalls für Überladungsapplikationen vollzogen.

Fixpunktberechnungen von typabhängigen Funktionen, die wir mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ eingeführt haben, sowie alle in System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ definierten Typoperationen werden zur Übersetzungszeit berechnet.

Vergleichen wir mit C++ . Typabhängige Terme (Wert- und Typfunktionen) entsprechen Klassentemplates, die Typapplikation der Template-Instanziierung und die mit `typecase` und `Typecase` verbundene *case*-Analyse der Template-Spezialisierung. Da Templates vor der Laufzeit expandiert werden müssen, folgt, daß auch in C++ alle typabhängigen Berechnungen zur Übersetzungszeit durchgeführt werden – das gilt auch für rekursive Templates, so daß man im Allgemeinen nicht garantieren kann, daß ein C++ -Übersetzer terminiert.

Projektionen entsprechen in C++ dem Zugriff auf die Komponenten eines Objekts. Die par-

tielle Auswertung der Projektion ist am ehesten mit der ***lightweight object optimization***[9] vergleichbar.

In der Regel werden Objekte in der Zielmaschine als Zeiger gespeichert und um auf ein Datenfeld des Objekts zuzugreifen, muß dieser Zeiger um ein entsprechendes Offset inkrementiert werden. Mit diesem indirekten Speicherzugriff gehen jedoch zahlreiche Optimierungsmöglichkeiten verloren. Bei der *lightweight object optimization* werden Feldwerte, die sich in einem Maschinenregister speichern lassen, zunächst in temporäre Variablen kopiert. Dadurch werden weitere Optimierungen, wie z.B. Registerzuweisung, oder *copy-* und *constant-propagation* möglich und der indirekte Zugriff ist nur initial, beim Laden eines Maschinenregisters erforderlich.

Die von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ zusätzlich durchgeführten ein-Schritt-β-Reduktionen würde einem *inlining*[10] in C++ entsprechen. Zwar kann man Funktionen mit dem Schlüsselwort `inline` markieren, jedoch gilt dies lediglich als Empfehlung an den Übersetzer. Ein *inlining* von mit `inline` markierten Funktionen ist nicht garantiert.

5.4 Bewertung der Template-Metaprogrammierung

Eine gerade für die Implementierung von Interpretern interessante Eigenschaft der C++ Template-Metaprogrammierung ist die, daß sie nicht am **Erweiterbarkeitsproblem** [31, 93, 182] leidet (manchmal spricht man auch vom *expression problem*).

Das Erweiterbarkeitsproblem beschreibt den Umstand, daß es in den meisten Programmiersprachen nicht ohne größeren Aufwand möglich ist, einen Datentyp und mit ihm gleichzeitig die Operationen, die auf seinen Instanzen operieren, zu erweitern.

Betrachten wir dieses Problem am Beispiel der Implementierung eines Interpreters, der auf einem AST operiert.

In funktionalen Sprachen läßt sich ein AST durch einen algebraischen Datentyp modellieren. Will man nun die interpretierte Sprache erweitern (z.B. durch Hinzunahme eines neuen Operators), so wird man den algebraischen Datentyp ebenfalls erweitern müssen. Gerade das ist in den meisten funktionalen Sprachen nicht so einfach möglich und zieht die Anpassung zahlreicher Funktionen nach sich, da diese jetzt auch mit dem neu hinzugekommenen Konstruktor zurecht kommen müssen.

Beabsichtigt man hingegen den Interpreter um eine neue Funktion (z.B. eine neue Optimierungsstufe) zu ergänzen, so ist dies in der Regel problemlos machbar — man schreibt einfach eine entsprechende Funktion.

In objektorientierten Sprachen ist es genau andersherum. Die Knoten des AST werden meist in einer Klassenhierarchie angeordnet, wobei alle Knoten eine gemeinsame Vaterklasse (eine gemeinsame Schnittstelle) haben. Unter Ausnutzung des Vererbungsmechanismus läßt sich diese Klassenhierarchie geradezu auf triviale Art und Weise erweitern – die Hinzunahme von neuen syntaktischen Konstrukten ist daher unproblematisch. Das Hinzufügen von Funktionen erweist sich jedoch als schwierig. Operationen auf einem AST werden in objektorientierten Sprachen in der Regel durch Anwendung des *visitor patterns* realisiert ([52, Seite 301]). Ergänzt man die Klassenhierarchie um einen neuen Knotentyp, muß man alle existierenden *visitor*-Klassen entsprechend anpassen – ein sehr aufwendiges und fehleranfälliges Unterfangen.

[9] Man spricht auch vom *scalar replacement of aggregates* – siehe [119, S. 331 ff.]

[10] Auch *procedure integration* genannt – siehe [119, S. 465 ff.].

Implementiert man einen Interpreter durch strukturelle Typanalyse (also unter Ausnutzung von Template-Metafunktionen), so hat man diese Probleme nicht. Der *pattern matching* Mechanismus von C++ erlaubt das einfache Hinzufügen von neuen Fällen; d.h. man kann jederzeit neue, oder sogar speziellere Muster zu einer Funktion hinzufügen. Da Template-Metafunktionen auf beliebigen Konstruktoren operieren können, ist die Hinzunahme eines neuen Konstruktors jederzeit möglich. Schließlich bereitet auch das Hinzufügen neuer Template-Metafunktionen keinerlei Schwierigkeiten.

Leider birgt die Template-Metaprogrammierung aber auch viele Nachteile, was nicht verwunderlich ist, da die Designer von C++ diese Programmiertechnik beim Entwurf von C++ nicht im Visier hatten.

Der *pattern matching* Mechanismus, der sowohl bei der Überladung von Funktionen, als auch bei der Template-Instanziierug zum Einsatz kommt, ist aufwendig und wurde von den meisten Compilerherstellern in seiner Bedeutung unterschätzt, so daß es kaum effiziente Implementierungen gibt. Als Konsequenz ergeben sich gerade bei großen Metaprogrammen sehr lange Laufzeiten (bzw. Übersetzungszeiten, da Metaprogramme ja während der Übersetzung eines C++ -Programmes ausgeführt werden).

Jede Typapplikation bedeutet die Instanziierung eines neuen Templates. Templates werden während des Übersetzungsprozesses jedoch nicht verworfen, so daß jeder Reduktionsschritt unwiderruflich Speicherplatz verbraucht. Hinzu kommt, daß einmal instanziierte Templates von vielen Übersetzern wie Spezialisierungen behandelt werden, so daß das *pattern matching* mit steigender Zahl der Instanzen immer langsamer wird.

Derzeit verfügbare C++ -Übersetzer führen keine Optimierungen für Template-Metafunktionen durch, so daß man insgesamt sowohl ein schlechtes Laufzeitverhalten, als auch einen hohen Speicherbedarf zur Ausführung von Template-Metaprogrammen beobachten kann.

Neben Effizienzproblemen, ist das Lesen von Template-Code sehr gewöhnungsbedürftig und erfordert einige Übung. Das Entwickeln von Template-Code ist aber auch mangels Möglichkeiten zum Debuggen schwierig. Selbst einfachste Debugging-Techniken, wie die Ausgabe von Meldungen auf dem Bildschirm, sind nicht möglich.

Mit der Möglichkeit, rekursive Typfunktionen implementieren zu können, geht in C++ , wie in System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ die Entscheidbarkeit der Typisierbarkeit eines Terms verloren. Im Anhang B des C++ -Standards [3, Seite 712] wird empfohlen, daß ein C++ -Übersetzer mindestens 17 rekursiv verschachtelte Template-Instanziierungen unterstützen sollte. Viele Compiler-Hersteller haben darin jedoch eine Obergrenze gesehen, um die Terminierung des Compilationsvorganges garantieren zu können – eine Limitierung der Rekursionstiefe bedeutet natürlich den Verlust der Berechnungsuniversalität auf der Typebenene.

Mit zunehmend komplexeren Programmbibliotheken zeichnete sich jedoch schnell ab, daß Handlungsbedarf besteht und so bieten die heute populärsten C++ -Übersetzer spezielle Optionen um die Verschachtelungstiefe von Template-Instanziierungen zu steuern, oder jegliche Beschränkung fallen zu lassen. Es bleibt jedoch die Empfehlung des C++ -Standards, so daß man sich streiten kann, ob der Typechecker eines standardkonformen C++ -Übersetzers terminieren muß, oder nicht (bzw. ob Typprogramme in der Praxis berechnungsuniversell sind).

Trotz dieser Widrigkeiten erfreut sich die Template-Metaprogrammierung einer wachsenden Anwenderschar und wird in zahlreichen C++ -Bibliotheken erfolgreich eingesetzt. Beispiele finden sich in den Bereichen High-Performance Computing [171, 91, 152, 98], generative Programmierung [42], Parser-Generatoren [60] und funktionale Programmierung [157, 80, 111].

5.5 Weiterführende Literatur

Der Begriff **Template-Metaprogrammierung** geht vermutlich zurück auf Todd Veldhuizen [170]. Er war auch der erste, der erkannte, daß sich der C++ -Template-Mechanismus als partieller Auswerter einsetzen läßt, um z.B. C++ -Ausdrücke zu optimieren [169][171].

Aktuell verfügbare Einführungen in die Template-Metaprogrammierung beschäftigen sich in erster Linie mit praktischen Beispielen (siehe z.B. [42, Kapitel 10], [168, Kapitel 17 und 18] und [156]).

Eine Arbeit, die die Grundkonzepte dieser Programmiertechnik losgelöst von C++ erörtert, ist uns nicht bekannt.

Teil II

Anwendung der Sprachintegration durch strukturelle Typanalyse:

Multiparadigma-Programmierung mit C++ durch Einbettung einer nicht-strikten funktionalen Sprache mit algebraischen Datentypen

Kapitel 6

Der Testfall: Multiparadigma-Programmierung

In seiner ursprünglichen Bedeutung steht das aus dem Griechischen stammende Wort *Paradigma* für „Beispiel" oder „Muster". Thomas Kuhn weitet diese Bedeutung in [97] deutlich aus. Er sieht in einem Paradigma ein Denkmuster, das die wissenschaftliche Orientierung einer Zeit prägt und definiert wissenschaftlichen Fortschritt als das Ersetzen eines Paradigmas durch ein neues.

In der Informatik wurde der Begriff vermutlich durch Robert W. Floyd eingeführt, der in seiner Turing Award Lecture von 1979 über *The Paradigms of Programming* referierte [47]. Floyd beschreibt drei Kategorien von Programmierparadigmen: feingranulare Programmiertechniken (z.B. Rekursion, Iteration, *call by name*, *call by value*, usw.), Methoden des algorithmischen Entwurfs (z.B. *divide and conquer*, dynamische Programmierung) und grobgranulare Programmiertechniken (z.B. funktionale und rule-based Programmierung).

Heute ist der Begriff des Programmierparadigmas hoffnungslos überladen. In der Literatur finden sich unter anderem: imperatives, objektorientiertes, funktionales und logisches Paradigma [26, 144, 71, 143, 107]; sowie *asynchronous*, *synchronous*, *transformational*, *form-based*, *dataflow*, *constraint*, *demonstrational* [8], *relational* [71, 143, 81], *lambda-free*, applicative, procedural [143, 81], subject-oriented [68] und *aspect-oriented paradigm* [92]; schließlich werden auch noch das generische und generative Programmierparadigma [42] genannt.

Eine genaue Definition und Abgrenzung der einzelnen Paradigmen wird durch ihre stetig wachsende Zahl nahezu unmöglich. Die größte Akzeptanz scheinen die vier zuerst genannten Paradigmen, also das imperative, das funktionale, das logische und das objektorientierte Paradigma, zu genießen, was man daraus schließen kann, daß sie gewissermaßen die Schnittmenge der von den verschiedenen Autoren genannten Paradigmen bilden.

Ein Programmierparadigma beschreibt eine Programmiermethodik und legt fest, welche Eigenschaften eine Sprache hinsichtlich Syntax, Semantik und Typsystem haben muß, um diese zu unterstützen. Wie wir im nächsten Abschnitt sehen werden, sind die Grenzen zwar nicht immer scharf, zum Teil ergeben sich aber auch sehr deutliche Unterschiede.

Gerade diese Unterschiede machen Multiparadigma-Programmierung zu einem interessanten Testfall, um die Grenzen einer Technik zur Spracheinbettung auszuloten.

Im nächsten Abschnitt werden wir die zentralen Eigenschaften der vier gängigsten Paradigmen kurz vorstellen. Dabei werden wir dem funktionalen Paradigma besondere Aufmerksamkeit

schenken, um noch fehlende Begriffe einzuführen, die wir bei der Beschreibung der Integration einer funktionalen Sprache in C++ benötigen.

6.1 Programmierparadigmen

6.1.1 Imperative Programmierung

Das imperative Paradigma reflektiert die Arbeitsweise der heute vorherrschenden Rechnerarchitektur des *von Neumann Rechners*. Ein imperatives Programm besteht aus einer Reihe von Anweisungen, die sequentiell ausgeführt werden und Einfluß auf die Werte von Variablen nehmen. Variablen selbst sind im Grunde genommen andere Namen (Aliase) für die Speicherzellen der Maschine.

Sofern sie das Konzept der Funktion unterstützen, werden imperative Sprachen auch *prozedurale* Sprachen genannt. Die Anwendung einer Funktion hat, im Gegensatz zum mathematischen Verständnis, zwei Effekte: Zum einen wird ein Wert an den Aufrufer zurückgegeben (funktionaler Anteil), zum anderen nehmen die Anweisungen der Funktion Einfluß auf den Zustand der Maschine – man spricht vom *Seiteneffekt* der Funktion.

Zentrale Konzepte von imperativen Sprachen sind Zuweisungen (insbesondere kann der Wert von Variablen mehrfach geändert werden), Schleifenkonstrukte (z.B. `for` und `while`) und Sprungbefehle (z.B. `goto` und `break`).

Beispiele für imperative Sprachen sind Pascal [1], C [4], Fortran [2], sowie die meisten Assemblersprachen.

6.1.2 Funktionale Programmierung

Eine klassische funktionale Programmiersprache könnte man als eine imperative Sprache beschreiben, in der es keine Variablen (als Aliase von Speicheradressen), keine Mehrfachzuweisungen und keine imperativen Kontrollstrukturen (Sprungbefehle, Schleifen) gibt. Im Zentrum des Interesses steht der Begriff der Funktion. Funktionen sind *first-class citizens* einer funktionalen Programmiersprache; das heißt, sie können sowohl als Argument an andere Funktionen übergeben werden, als auch selbst Funktionsergebnis sein. Solche Funktionen nennt man auch **Funktionen höherer Ordnung**. Ein klassisches Beispiel ist die Funktionskomposition $\circ$: $(f \circ g)(x) = f(g(x))$.

Funktionale Programme entstehen durch funktionale Komposition; Rekursion ist dabei ein zentrales Konzept. Syntaktisch gesehen besteht ein funktionales Skript aus einer Reihe von Funktionsgleichungen. Ein Programm besteht aus einem Skript und einem Ausdruck, welcher anhand der Gleichungen des Skripts reduziert wird.

Syntaktisch gesehen gibt es in funktionalen Sprachen nur unäre Funktionen. Dadurch sind sie aber nicht weniger ausdruckskräftig - im Gegenteil! Die Stelligkeit einer Funktion ist eine rein syntaktische Eigenschaft. Mathematisch gesehen, kann man sich eine n-stellige Funktion als eine Funktion vorstellen, die auf *einem* n-Tupel operiert - zurückübertragen auf die syntaktische Sicht, wäre sie dann einstellig (unär).

Currying bringt die beiden Sichtweisen in Einklang. Durch currying wird es möglich, auf kartesische Typen zu verzichten, was Schönfinkel um ca. 1920 als erster gezeigt hat - man

spricht auch von der **Curry-Version** einer Funktion. Der Trick liegt in der Verwendung von Funktionen höherer Ordnung:

Gegeben sei $f : A_1 \times \cdots \times A_n \to R$. Dann kann man die Curry-Version zu f,

$$f' : A_1 \to \cdots \to A_n \to R$$

wie folgt konstruieren, so daß $f'(a_1) \cdots (a_n) = f(a_1, \cdots, a_n)$ gilt:

$$\begin{aligned} f'(a_1) &= g_{a_1} \\ f'(a_1)(a_2) &= g_{a_1,a_2} \\ &\vdots \\ f'(a_1)(a_2) \cdots (a_{n-1}) &= g_{a_1,\cdots,a_{n-1}} \\ f'(a_1) \cdots (a_n) &= f(a_1, \cdots, a_n) \end{aligned}$$

mit

$$\begin{aligned} g_{a_1}(a_2) &= f'(a_1)(a_2) \\ g_{a_1,a_2}(a_3) &= f'(a_1)(a_2)(a_3) \\ &\vdots \\ g_{a_1,\cdots,a_{n-2}}(a_{n-1}) &= f'(a_1) \cdots (a_{n-2})(a_{n-1}) \end{aligned}$$

Aus syntaktischer Sicht ist f' ein unäres Funktionssymbol. Aus der Sicht des Typsystems ist f' eine Funktion, die ein Argument vom Typ A_1 übernimmt und eine unäre Funktion zurückliefert, die ein Argument vom Typ A_2 übernimmt und eine unäre Funktion zurückliefert, die ein Argument vom Typ A_3 übernimmt und eine unäre Funktion zurückliefert $\cdots$, die ein Argument vom Typ A_n übernimmt und einen Wert vom Typ R zurückliefert.

Currying bedeutet aber mehr als den puren Verzicht auf Produkttypen. Der eigentliche Nutzen ergibt sich erst im Zusammenspiel mit Funktionen höherer Ordnung. Man kann f' wahlweise mit einem bis n Argumenten aufrufen. Ruft man f' mit weniger als n Argumenten auf, so spricht man von **partieller Applikation**. Die dabei entstehende Funktion g_i ist gewissermaßen eine Spezialisierung von f, in der einige Variablen durch konkrete Werte ersetzt worden sind - g_i „merkt" sich sozusagen die Parameter, die bereits an f' übergeben worden sind.

Funktionen können in funktionalen Sprachen *on-the-fly* durch so genannte **λ-Ausdrücke** generiert werden. Zum Beispiel erzeugt der Ausdruck $\lambda x.\lambda y.x + y$ eine namenlose Funktion zur Addition zweier Argumente. λ-Ausdrücke haben ihren Ursprung im λ-Kalkül, der Grundlage für die Semantik funktionaler Sprachen (zum λ-Kalkül siehe [13]).

Weitere Merkmale, die funktionale Sprachen typischerweise aufweisen, sind: algebraische Datentypen, pattern matching und die **verzögerte Auswertung** (auch **lazy evaluation** oder *call by need*- bzw. *leftmost outermost* Reduktionsstrategie genannt). Wird eine Funktion mit einem Ausdruck als Argument aufgerufen, wird dieser nicht sofort ausgewertet (**eager evaluation** oder *call by value* bzw. *leftmost innermost* Reduktionsstrategie), sondern erst dann, wenn sein Wert auch tatsächlich gebraucht wird; daher der Name *verzögerte* Auswertung.

Bei einfacher Umsetzung kann sich die verzögerte Auswertung von Funktionsargumenten negativ auf die Laufzeiteffizienz eines Programmes auswirken. Hierzu ein Beispiel:

```
square x = x * x
test y = square (square y)
```

Betrachten wir die Auswertung des Ausdrucks `test 3`[1]. Gemäß der zweiten Gleichung kann dieser zu `square (square 3)` reduziert werden. Weitere Reduktion ist durch Anwenden der ersten Gleichung möglich: Der Ausdruck `square 3` wird an den Parameter `x` gebunden - essentiell für die verzögerte Auswertung ist ja, daß `square 3` an dieser Stelle gerade **nicht** reduziert wird. Substituieren von `x` durch `square 3` in der linken Seite der ersten Gleichung ergibt:

```
(square 3) * (square 3)
```

Im Gegensatz zur *call by value* Auswertung wird der Ausdruck `square 3` hier zweimal ausgewertet.

Graphreduktion [175] ist eine spezielle Reduktionsstrategie, die das doppelte Auswerten von Funktionsargumenten zu vermeiden hilft. Es handelt sich hierbei um einen verhältnismäßig komplexen Auswertungsmechanismus, der in den meisten Fällen durch eine abstrakte Maschine modelliert wird (Template Instantiation Machine [140], G-Machine [82, 9], Spinless Tagless G-Machine [135] etc.).

Funktionale Programme werden vorwiegend zunächst in den Maschinencode (Bytecode) einer dieser abstrakten Maschinen übersetzt. Das dabei entstandene Maschinenprogramm wird dann von einem Emulator (einem Bytecode-Interpreter) abgearbeitet, in die Maschinensprache eines realen Rechners übersetzt, oder aber auf einer Spezialhardware ausgeführt.

Funktionale Sprachen verfügen über sehr komplexe Typsysteme. Parametrische Polymorphie etwa war als erstes in der Sprache ML [132] zu finden. Moderne funktionale Sprachen wie Haskell [136] ermöglichen darüber hinaus das Überladen von Operatoren und Funktionen. Typisch für funktionale Sprachen ist, daß Typen vom Compiler inferiert werden; d.h. der Benutzer muß Variablen und Funktionen nicht mit Typannotationen versehen.

Man unterscheidet pure und nicht-pure funktionale Sprachen, je nachdem, ob Funktionen frei von Seiteneffekten sind, oder nicht. Zu den puren funktionalen Sprachen gehören beispielsweise Haskell [136], Miranda [70] und pure Lisp. ML [132] und Scheme [49] gehören zu den nicht-puren funktionalen Sprachen.

6.1.3 Logische Programmierung

Wie die funktionalen Sprachen gehören logische Programmiersprachen zu den *deklarativen* Sprachen.

Ein logisches Skript beinhaltet eine Reihe von Fakten und Regelsätzen (Implikationen, oder auch *predicates*). Ein logisches Programm besteht neben einem Skript aus einem Ausdruck (dem *goal*). Während der Ausführung eines Programmes wird versucht, das *goal* mit Hilfe der Fakten und Regeln des Skripts zu beweisen. Enthält das *goal* Variablen, so wird jede mögliche

[1] Wir setzen hier ein grundlegendes Verständnis von Termersetzungssystemen voraus. Einen guten Einstieg vermittelt [11].

Belegung für diese Variablen gesucht, die das *goal* unter Berücksichtigung des Skripts wahr werden lassen.

Die Ausführung eines logischen Programmes entspricht einem Suchvorgang, bei dem u.U. mehrere Alternativen geprüft werden müssen. Und in der Tat spiegelt das Ausführungsmodell dieser Sprachen üblicherweise die SLD-Resolution der Prädikatenlogik wider [148].

Die erste und wohl auch bekannteste logische Programmiersprache ist PROLOG (PROgramming in LOGic). PROLOG hat ein monomorphes Typsystem, der Benutzer muß keine Typannotationen vornehmen [154].

Eine abstrakte Maschine zur Ausführung von PROLOG-Programmen, die auf den Prinzipien der SLD-Resolution aufbaut, ist die *Warren Abstract Machine* (WAM)[176]. Ähnlich wie funktionale Sprachen mit verzögerter Auswertung, werden PROLOG Programme zunächst in den Bytecode einer abstrakten Maschine übersetzt.

6.1.4 Objektorientierte Programmierung

Die objektorientierte Programmierung ist orthogonal zu den drei oben genannten Paradigmen zu sehen. Objektorientierte Sprachen sind meist Erweiterungen von imperativen (C++ [3], JAVA), funktionalen (O'Haskell [126], OCaml) oder logischen Sprachen (SPOOL[50]).

Obwohl Objektorientierung in der Praxis als Erweiterung der anderen Paradigmen auftaucht, muß man sie als eigenes Paradigma verstehen. Im Verlauf des Softwareentwicklungsprozesses wirken sich die unterschiedlichen Paradigmen nämlich nicht erst zur Implementierungsphase aus, sondern bereits viel früher, in der Entwurfsphase. Beim objektorientierten Entwurf spielen funktionale und imperative Aspekte eine eher untergeordnete Rolle. Im Zentrum des Interesses stehen Klassen und Zusammenhänge von Klassen bzw. die Interaktion von Objekten [78].

Wir der Name schon sagt, basieren objektorientierte Programmiersprachen auf dem Konzept des Objekts. Ein Objekt ist ein abstrakter Datentyp, der einen Zustand (die *Datenfelder*) des Objekts und eine Schnittstelle in Form von Prozeduren und Funktionen (den *Methoden*) zur Verfügung stellt, um seinen Zustand zu manipulieren oder sonstige Aktionen auszuführen.

Eine zentrale Idee bei der objektorientierten Programmierung besteht darin, Objekte, die Gemeinsamkeiten in ihren Schnittstellen aufweisen, unter bestimmten Voraussetzungen als typgleich zu behandeln.

Am populärsten ist die Methode, gemeinsame Schnittstellenteile in einer Klasse zu definieren. Eine Klasse ist ein Typ und je nach Programmiersprache kann die Zugehörigkeit eines Objekts zu einer Klasse sich allein an seiner Schnittstelle orientieren (strukturelle Korrespondenz), oder aber explizit vom Programmierer festgelegt werden (namentliche Korrespondenz).

Klassen können hierarchisch organisiert sein. Eine neue Klasse kann die Vereinbarungen einer bestehenden Klasse übernehmen (erben); gegebenenfalls können neue Methoden und Datenfelder hinzugefügt und die Implementierung bestehender Methoden angepaßt (überschrieben) werden. Ist eine Klasse `K` von einer Klasse `V` abgeleitet worden, so nennt man `K` **Kindklasse** und `V` **Vaterklasse** oder **Superklasse** und schreibt `K <: V`.

Der Vererbungsmechanismus garantiert, daß Objekte, die der Kindklasse angehören mindestens dieselbe Schnittstelle aufweisen, wie es Objekte der Vaterklasse tun. Objekte einer Kindklasse können damit als Stellvertreter für Objekte einer ihrer Vaterklassen eingesetzt werden. Die Vielgestaltigkeit (Polymorphie) eines Objekts liegt in der Klassenhierarchie begründet, die

```
# include <iostream>
using namespace std;

class V {
public:
  virtual void m() {
    cout << "V's test." << endl;
  }
};
class K : public V { // K ist Kindklasse von V
public:
  virtual void m() { // Methode m wird überschrieben
    cout << "K's test." << endl;
  }
};

void f(V* v) {
  v->m(); // <- dynamischer dispatch, basierend auf
          //     tatsächlichem Typ von v.
}

int main(int argc,char** argv) {
    K* k = new K(); // erzeuge Objekt vom Typ K
    f(k);           // <- Ausgabe von "K's test."
}
```

Abbildung 6.1: Überschreiben von Methoden / *dynamic dispatch* am Beispiel der Programmiersprache C++ .

im Grunde eine Inklusionsbeziehung definiert. Daher spricht man in diesem Zusammenhang auch von **Inklusionspolymorphie** [5].

Ein mit Inklusionspolymorphie und dem Überschreiben von Methoden in enger Verbindung stehender Mechanismus, der ebenfalls typisch für objektorientierte Sprachen ist, ist das sogenannte *dynamische binden* (auch *dynamic dispatch* genannt): Beim Aufruf einer Methode soll immer der tatsächliche Typ eines Objektes entscheiden, welche Methode aufgerufen wird. Ohne diesen Mechanismus macht das Überschreiben von Methoden eigentlich wenig Sinn.

Betrachten wir ein Beispiel (siehe Abbildung 6.1). Wird in einer von `V` abgeleiteten Kindklasse `K` die Methode `m` überschrieben und wird ein Objekt vom Typ `K` an eine Funktion `f` übergeben, die ein Objekt vom Typ `V` erwartet, so wird ein Aufruf von `m` aus `f`, sich auf die überschriebene Version von `m` beziehen. Das heißt, der Laufzeittyp ist entscheidend dafür, welche Methode aufgerufen wird.

Objektorientierte Sprachen sind inhärent polymorph (Inklusionspolymorphie). Es gibt aber auch objektorientierte Sprachen, die darüber hinaus parametrische Polymorphie unterstützen (z.B. PIZZA [128] und C++ [158]) und/oder das Überladen von Funktionen, Methoden (JAVA [57], C++) und Operatoren (C++) erlauben.

Weitere Beispiele für objektorientierte Sprachen (OO Sprachen) sind Smalltalk [180], und Eiffel [113].

6.2 Multiparadigma-Programmierung

Abgesehen von der „Zwittergestalt" objektorientierter Sprachen kann die Zugehörigkeit einer Programmiersprache zu *einem* Paradigma in der Regel recht klar beantwortet werden; obwohl es auf feingranularer Ebene durchaus Überschneidungen gibt. Zum Beispiel nimmt der `cut` Operator der Programmiersprache PROLOG Einfluß auf den Kontrollfluß eines Programmes und kann daher als imperatives Sprachmerkmal verstanden werden.

In funktionalen Sprachen wie Haskell [136] oder Gofer wird das Konzept der Monaden [115, 173, 141] unterstützt. Monaden erlauben die logisch konsistente Integration von Berechnungen mit Seiteneffekten in pure funktionale Sprachen und werden daher gerne zur Realisierung von Ein- / Ausgabe-Operationen eingesetzt. Der monadische Programmierstil ist eher imperativer Natur, was an Haskells `do`-Notation besonders gut zum Ausdruck kommt:

```
- Eine do-Anweisung umschließt mehrere Ausdrücke,
- die sequentiell abgearbeitet werden:
main = do putStr ''Bitte geben Sie Ihren Namen ein: ''
          x <- getLine
          putStr ''Ihr Name ist: '' ++ x ++ ''\n''
          return ()
```

Derart feingranulare Gesichtspunkte machen aber weder PROLOG noch Haskell oder Gofer zu Multiparadigma-Sprachen. Wir wollen eine Sprache erst dann als Multiparadigma-Sprache ansehen, wenn sie die grobgranularen Aspekte mehrerer Paradigmen in sich vereint; dazu gehört etwa, daß sie die zugehörigen Design- und Entwurfsmethoden mit Sprachmitteln hinreichend unterstützt, dazu gehört insbesondere aber auch ihr Ausführungsmodell (ihre operationelle Semantik) und ihr Typsystem (polymorph vs. monomorph, algebraische Datentypen, Klassen etc.). Nicht zu vernachlässigen ist die unmittelbare Schnittstelle einer Programmiersprache zum Programmierer: die Syntax einer Sprache.

Wir wollen dann von **Multiparadigma-Programmierung** sprechen, wenn innerhalb eines Softwareprojektes mehrere Paradigmen zum Einsatz kommen.

Hinsichtlich der Einbettung von Programmiersprachen bedeutet die Multiparadigma-Programmierung eine besondere Herausforderung.

Da sich die Sprachen der einzelnen Paradigmen hinsichtlich Auswertungsmechanismus und Typsystem z.T. sehr stark voneinander abgrenzen, sind die Anforderungen an die Hostsprache besonders groß. Zum Beispiel muß man bei der Einbettung einer funktionalen Sprache mit verzögerter Auswertung in eine imperativ-objektorientierte Sprache, eine von der Hostsprache abweichende Auswertungsstrategie implementieren. Imperativ-objektorientierte Sprachen kennen keine algebraischen Datentypen, so daß man ein entsprechendes Typsystem durch die Hostsprache simulieren können muß.

Kapitel 7

Die funktionale Sprache EML

In den nächsten Kapiteln werden wir die Einbettung einer funktionalen Programmiersprache in C++ diskutieren. Dabei werden wir auf die Techniken zurückgreifen, die wir im Rahmen der Integration von **V** in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ (siehe Abschnitt 4.2.4) vorgestellt haben.

Die einzubettende Sprache nennen wir **EML** (*Embedded Meta-Language*).

In diesem Kapitel wollen wir einen Überblick (einen ersten Eindruck) von EML vermitteln. Dazu werden wir Syntax, Semantik und Typsystem dieser Sprache anhand von Beispielen vorstellen. Eine formale Definition von EML reichen wir in den folgenden Kapiteln nach.

7.1 Syntax von EML

Ein Design-Ziel von EML war die syntaktische Nähe zu etablierten funktionalen Sprachen wie ML, Gofer oder Haskell. Wie wir sehen werden, müssen wir aufgrund der syntaktischen Einbettung in C++ an einigen Stellen Kompromisse eingehen.

Ein EML-Skript (siehe Abbildung 7.1) besteht aus einer Reihe von Funktionsgleichungen der Form $s(x_1)\cdots(x_n) = e$, die durch Kommata voneinander zu trennen sind. s und x_1 bis x_n sind Variablen, e ist ein EML-Term. Die Variable s nennen wir **Superkombinator** und sprechen bei obiger Gleichung auch auch von einer Superkombinatordefinition.

```
divisible(x)(y) = x % y == 0                , // 1
forward(x)      = 2*bar()                   , // 2
twice(f)(x)     = f(f(x))                   , // 3
average(x)(y)   = (x+y) / 2                 , // 4
fac(c)          = IF(n == 0,1,n*fac(n-1))   , // 5
foo(x)          = -twice(average(x), x)     , // 6
bar()           = LET[ x=twice(foo(1)),       // 7
                       y=10+x
                     ]( x + y ),
diverge(x)      = diverge(x)                  // 8
```

Abbildung 7.1: Beispiel für ein EML-Skript

Im Vergleich zu Haskell müssen Funktionsparameter und -argumente mit Klammern umschlossen werden. Anstelle von `divisible x y` schreibt man in EML `divisible(x)(y)`. Ausgenommen hiervon sind die Vorzeichenoperatoren `+` und `-`, sowie der Negationsoperator `!` und binäre Operatoren (`+`, `-`, `==` usw.). Diese dürfen ausschließlich in der präfix- bzw. infix-Notation verwendet werden.

Parameterlose Funktionen (z.B. `bar` in Zeile 7) sind mit einem runden Klammerpaar zu versehen, wenn sie definiert oder aufgerufen werden. Superkombinatordefinitionen dürfen rekursiv sein (siehe Funktion `diverge`).

`IF` stellt sich in EML als dreistellige Funktion dar, allerdings ist man hinsichtlich der Klammerung der Argumente flexibler, um partielle Applikationen besser zu unterstützen: `IF(t)(t)(t)`, `IF(t,t)(t)` und `IF(t,t,t)` sind allesamt gültige Schreibweisen. Wie üblich, ist das erste Argument an `IF` eine zu prüfende Bedingung (i.d.R. ein boolescher Ausdruck); das zweite Argument wird zurückgegeben, wenn die Bedingung sich zur Konstanten `true` auswerten läßt, ansonsten das dritte. `IF` entspricht damit dem Fragezeichenoperator `operator?:` in C++ , der jedoch nicht überladen werden kann, so daß wir hier auf eine spezielle Syntax zurückgreifen müssen.

Funktionen liegen in EML grundsätzlich in der Curry-Version vor und können somit partiell appliziert werden (siehe z.B. `average(x)` in Zeile 6 von Abbildung 7.1).

Der Sichtbarkeitsbereich von Funktionsnamen erstreckt sich über das gesamte Skript. Eine Funktion kann daher aufgerufen werden, bevor sie definiert wurde (z.B. kann die Funktion `bar` bereits in Zeile 2 verwendet werden, obwohl sie erst in Zeile 7 definiert wird).

Zum Einführen von lokal sichtbaren Variablen stellt EML `Let`-Ausdrücke zur Verfügung. Ein `Let`-Ausdruck besteht aus zwei Komponenten: Einer Liste von Gleichungen, die jeweils einer Variablen einen Term zuordnen, und einem EML-Term, in dem diese Variablen verwendet werden können. Die Gleichungsliste muß von eckigen Klammern umschlossen werden und die einzelnen Gleichungen sind durch Kommata voneinander zu trennen.

EMLs `Let`-Ausdrücke entsprechen den `Let`-Ausdrücken, die wir auf Seite 47 für System $\mathbf{F}_\omega$ eingeführt haben.

Der Sichtbarkeitsbereich einer Variablen erstreckt sich über alle nachfolgenden Variablendefinitionen und natürlich den EML-Term. Die Verwendung der Variable `x` in der Definition von `y` ist also zulässig.

Ein EML-Programm besteht aus einem EML-Skript und einem Ausdruck, der entsprechend den Funktionsgleichungen des Skripts ausgewertet werden soll. Die Integration eines EML-Programmes in C++ stellt sich wie folgt dar:

```
int fac10 =
   ( fac(n) = IF(n==0, 1, n*fac(n-1)) )[ fac(10) ];

int fac20 = fac(20);  // FEHLER!
```

Das Skript wird von runden Klammern umschlossen und der auszuwertende Audruck in eckigen Klammern dahinter gesetzt.

Auf die Funktion `fac` kann nur vom EML-Skript und vom auszuwertenden EML-Term aus zugegriffen werden. Die zweite Programmzeile führt daher zu einem Übersetzungsfehler.

Grundsätzlich könnte man EML-Skripte in einem C++ -Programm auch global sichtbar machen. Wie wir noch noch sehen werden, wäre das für den Programmierer aber sehr aufwendig.

```
#include <complex>
struct MyVec {
  MyVec(int _x1,int _x2) : x1(_x1),x2(_x2) {}
  int x1,x2;
};

MyVec operator+(const MyVec& a,const MyVec& b) {
  return MyVec(a.x1+b.x1 , a.x2+b.x2 );
}

void test() {
  MyVec a(2,3),b(4.5);
  std::complex<double> c;
  // ... Initialisierung der Vaiablen ...
  ( f(x)(y) = x + y )[ f(a)  (b) ];   // f:: MyVec -> MyVec -> MyVec
  ( f(x)(y) = x + y )[ f(1.3)(2) ];   // f:: double -> int -> double
  ( f(x)(y) = x + y )[ f(a)  (c) ];   // FEHLER!
                                      // operator+(MyVec,complex<double>)
                                      // hier nicht sichtbar!
}

MyVec operator+(const MyVec& a,const std::complex<double>& c) {
  return MyVec(a.x1 + c.real(), a.x2+c.imag() );
}
```

Abbildung 7.2: Typsystem von EML.

7.2 Typsystem von EML

Um einen möglichst schnellen und einfachen Wechsel von C++ zu EML und umgekehrt zu ermöglichen, ist das Typsystem von EML größtenteils kompatibel mit dem von C++ .

Wie in funktionalen Sprachen üblich, sind explizite Typannotationen in EML-Skripten nicht erforderlich. Jede Funktion ist zunächst polymorph – genau genommen ist sie beschränkt parametrisch polymorph.

Die Beschränkung ergibt sich allerdings nicht aus einer Typklasse, oder einer Obergrenze bezüglich einer Klassenhierarchie (so etwas gibt es in EML nicht), sondern aus der Bedeutung von Funktionen und Operatoren innerhalb des C++ -Typsystems.

Die in EML eingebauten Operatoren werden auf die entsprechenden C++ -Operatoren abgebildet. Zum Beispiel hängt die Bedeutung vom Operator `+` in einem EML-Skript vom C++ -Kontext ab, in dem dieses Skript steht (siehe Abbildung 7.2).

Innerhalb der EML-Funktion `f` sind jeweils alle Überladungen von `operator+` sichtbar, die sich im Sichtbarkeitsbereich der C++ -Funktion `test` befinden. Neben den vordefinierten Funktionen gehört dazu auch der für Objekte vom Typ `MyVec` überladene Additionsoperator. Die Funktion `operator+` zur Addition von `MyVec`-Objekten und komplexen Zahlen ist im EML-Skript jedoch nicht sichtbar und daher würde die Übersetzung des dritten EML-Programmes mit einem Typfehler abbrechen.

Einen allgemeinsten Typ für `f` zu finden ist nicht sinnvoll, da er die Einschränkungen, die

sich durch die vorhandenen Überladungen ergeben, nicht berücksichtigt. Betrachten wir zum Beispiel die möglichen Typen von `f`:

`unsigned int` → `unsigned int` → `unsigned int`

`signed int` → `signed int` → `signed int`

`unsigned char` → `unsigned char` → `unsigned char`

`signed char` → `signed char` → `signed char`

`double` → `double` → `double`

... weitere eingebaute Operatoren ...

`MyVec` → `MyVec` → `MyVec`

`MyVec` → `int` → `int`

Schließt man die in C++ möglichen Konvertierungen in die Betrachtung mit ein, so kommen noch zahlreiche Typen hinzu (z.B. `double` → `int` → `double`).

Das Typsystem von EML ist sehr flexibel. Trotz fehlender Typannotationen erlaubt EML die Definition polymorph-rekursiver Funktionen:

```
polyrec(x) = IF(x==0,
                1,                  // * -> int
                IF(x==1,
                   polyrec(0.5*x),  // double -> int
                   polyrec(x-1)     // int -> int
                  )
                )
```

Aus theoretischer Sicht mag dies zunächst verblüffend wirken, da Typinferenz unter Beteiligung von polymorpher Rekursion nicht entscheidbar ist. Dieses Problem ist jedoch nicht auf EML übertragbar, da Typinferenz hier nicht die Suche nach einem allgemeinsten Typ bedeutet, sondern vielmehr die Frage beantwortet, welchen Typ ein EML-Programm als Ergebnis liefert, wenn es mit konkreten Argumenten aufgerufen wird.

In EML können keine Datentypen definiert werden. Jeder C++ -Datentyp ist in EML verwendbar. Die EML-Umgebung stellt aber ein Rahmenwerk zur Verfügung, mit dem sich polymorphe algebraische Datentypen definieren lassen, so daß sie sowohl im funktionalen EML-, als auch im objektorientierten C++ -Umfeld einsetzbar sind (Details erläutern wir in Kapitel 11).

Ein algebraischer Listentyp ist in EML vordefiniert und stellt sich auf Seiten von C++ als Klassentemplate `TList<T>` dar, mit `T` als Platzhalter für einen beliebigen Elementtyp. `TList` ist kompatibel zum STL-Sequenzcontainertyp `std::list<T>`, so daß die Algorithmen aus der C++ -Standardbibliothek wiederverwendet werden können.

7.3 Semantik von EML-Programmen

EML-Programme werden nach dem *call by name* Prinzip ausgewertet. Das heißt, Argumente werden unausgewertet an eine Funktion übergeben und erst dann ausgewertet, wenn auf sie

```
TList<int> primes100  = (
    divisible(x)(y) = (x % y) == 0,

    factors(x)      = filter(divisible(x))(fromTo(1)(x)),
    isPrime(x)      = factors(x) == (cons(1)(cons(x)(nil()))),
    allPrimes()     = filter(isPrime)(from(1)),

    m(x)            = take(x)( allPrimes )
   )[ m(100) ];

cout << "Die ersten 100 Primzahlen sind: " << primes100 << endl;
cout << "Jetzt wird es schneller" << endl;
cout << primes100 << endl;
```

Abbildung 7.3: Primzahlberechnung in EML.

zugegriffen wird. Grundoperationen (z.B. `+`, `-`, `*` und `/`) sind, wie üblich, strikt in beiden Argumenten; die einfache Selektion `IF` ist nur im ersten Argument (der Bedingung) strikt.

Ähnlich wie bei der verzögerten Auswertung von Typfunktionen in C++ (siehe Seite 137) bekommen Funktionen, die Argumente verzögert auswerten, Funktionsabschlüsse (*closures*) als Argumente übergeben. Der Aufruf und die Abarbeitung einer Funktion ist daher weniger effizient, als bei einer strikten Auswertungsstrategie: Vor dem Aufruf muß Speicherplatz für den Abschluß reserviert werden und er muß entsprechend des Funktionsargumentes initialisiert werden. Darüber hinaus erfordert die Auswertung des Abschlusses selbst in der Regel einen indirekten Speicherzugriff und erschwert weitergehende Optimierungen (z.B. *common subexpression elimination* und *constant folding*).

Wird ein Argument in einer Funktion ohnehin in ausgewerteter Form benötigt, so ist dieser Mehraufwand überflüssig. In EML hat der Benutzer daher die Möglichkeit, die Parameter einer Superkombinatordefinition mit einer Striktheitsannotation zu versehen, um so die strikte Auswertung des zugehörigen Arguments zu erzwingen:

```
square(!x) = x * x
```

Da der eingebaute Multiplikationsoperator strikt in beiden Argumenten ist, wird der Wert von `x` in jedem Fall gebraucht. `square` ist somit strikt in `x`, was der Programmierer durch ein vorangestelltes Ausrufungszeichen zum Ausdruck bringen kann. Eine automatische Striktheitsanalyse bietet EML nicht.

Dank der verzögerten Auswertung kann man in EML mit unendlichen Listen programmieren. Zur Bearbeitung von Listen stehen eine Reihe von vordefinierten Funktionen zur Verfügung, die man auch in der Haskell-Bibliothek findet (z.B. `map`, `fold`).

Eine klassische Anwendung zur Demonstration des Programmierens mit unendlichen Listen ist die Berechnung von Primzahlen mit Hilfe des Siebes von Erathostenes (siehe Abbildung 7.3).

Als Folge der verzögerten Auswertung wird die Liste der ersten hundert Primzahlen erst während der Ausgabe berechnet. Die zweite Ausgabe wird daher flüssiger erscheinen.

In EML gibt es keine Möglichkeit zum *pattern matching*. Der Typ einer Konstruktorzelle muß explizit abgefragt werden. Hierzu stehen die eingebauten Testfunktionen `isNIL` und `isCONS` zur Verfügung. Zum Zerlegen einer Konstruktorzelle muß auf die Funktionen `head` und `tail` zurückgegriffen werden.

Eine Funktion zur Berechnung der Länge einer Liste sieht in EML daher so aus:

```
length(x) = IF( isNIL(x),
                0,
                1 + length( tail(x) )
              )
```

Innerhalb eines EML-Skriptes dürfen auch C++ -Ausdrücke verwendet werden. Diese werden vor der Abarbeitung des EML-Programmes ausgewertet. Betrachten wir hierzu ein Beispiel:

```
struct MyClass {
  MyClass(int _x) : x(_x) {}
  int m(int a) { return x*a; }
  double bm(int a,int b) { return a+b / x;}
  int x;
};

MyClass object(2);
// ...
int val = ( f(x) = x + object.m( 12 ) )[ f(2) ];
```

Der Ausdruck `object.m(12)` wird vor der Abarbeitung des EML-Skriptes zu 24 reduziert. EML-Skripte durchlaufen gewissermaßen einen partiellen Auswerter, wobei die *binding time analysis* sehr einfach ist: EML-Variablen sind grundsätzlich statisch, C++ -Ausdrücke (und damit auch Variablen) sind grundsätzlich dynamisch. Im Gegensatz zu klassischer partieller Auswertung erzwingen C++ -Funktionen und Methoden statische Argumente.

Das unmittelbare Weiterreichen von EML-Ausdrücken an C++ -Funktionen oder Methoden ist nicht erlaubt. Zum Beispiel würde die Verwendung des Terms `object.m( x+1 )` zu einem Typfehler führen, da der Term `x+1` zum Typ `int` inkompatibel ist.

Bevor man C++ -Funktionen und Methoden aus EML-Termen heraus aufrufen kann, muß man sie mit dem `call` Operator in EML-Funktionen umwandeln, die mit dynamischen Argumenten zurecht kommen und in Curry-Form vorliegen:

```
eml2cpp(x) = call(sin)(x) + call(object.bm)(x+2)(x/2)
```

Einzelheiten besprechen wir in Kapitel 10, Abschnitt 10.4.6.

Rekursion ist in EML das einzige Konzept, um Schleifen zu programmieren. Der EML-Übersetzer unterstützt die Optimierung von endrekursiven Aufrufen, sofern diese durch das spezielle Kommando `jump` eingeleitet werden:

```
fac(accu)(x) = IF(x==0,
                  accu,
                  jump(fac)(x*axxu)(x-1)
                 )
```

Kapitel 8

Syntax und syntaktische Einbettung von EML in C++

In diesem Kapitel geben wir eine formale Syntax für EML an und beschreiben die Übersetzung von EML in CoreEML, eine vereinfachte Kernsprache, in der es keine infix-Operatoren und keine Funktionsapplikationen höherer Ordnung gibt. Wir werden zeigen, wie man die Übersetzung von EML nach CoreEML durch geschicktes Ausnutzen von Überladung in C++ realisieren kann und wie sich der AST zu einem CoreEML-Programm unter Einsatz von C++ -Klassentemplates darstellen läßt.

8.1 Konkrete und abstrakte Syntax von EML

Die konkrete Syntax zu EML beschreiben wir, wie üblich, durch eine kontextfreie Grammatik (siehe Abbildung 8.1) Die Grammatik ist in folgendem Sinne unvollständig: Da wir EML in C++ einbetten, EML also eine Teilmenge von C++ ist, sind Operatorpräzedenzen und Assoziativitäten bereits fest vorgegeben. Ebenfalls unberücksichtigt bleibt die Möglichkeit Ausdrücke zu klammern.

Die Menge der funktionalen Variablen v entspricht einer Menge von C++ -Variablen, die sich durch einen ganz bestimmten Typ auszeichnen (dazu später mehr). Zur Abgrenzung von „normalen" Variablen sprechen wir manchmal auch von *funktionalen Variablen*.

Die Metavariable c repräsentiert C++ -Konstanten. Darin eingeschlossen sind numerische Konstanten, die Wahrheitswerte `true` und `false`, sowie Konstanten mit benutzerdefiniertem Typ. Wir werden später sehen, daß diese Menge der Konstanten noch etwas größer ist, da (wie bei der Integration von $\mathbf{V}$ in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$) EML und C++ -Terme gemischt werden können.

Streng genommen müßten wir an dieser Stelle von pre-EML sprechen, da die Grammatik die Produktion von semantisch fragwürdigen EML-Skripten zuläßt (zum Beispiel dürfen beliebige C++ -Konstanten als Funktionen verwenden werden, was natürlich unsinnig ist). Derart fehlerhafte Programme werden später durch einen semantischen Analyseschritt herausgefiltert.

Vor der Übersetzung nach C++ werden EML-Programme in eine einfachere Kernsprache übersetzt, die wir CoreEML nennen und in der es u.a. keine infix-Operatoren und keine spezielle syntaktische Form für `IF` gibt (siehe Abbildung 8.2). Darüber hinaus werden Funktionsapplikationen höherer Ordnung (z.B. die Anwendung der Funktion f auf die Funktion g) in

Syntax — EML

$t ::=$		**Terme:**
	v	*Funktionale Vairable*
	c	C++ `-Konstante`
	$t\ op_{Bin}\ t$	*Infix-Operatoren*
	$op_{Un}\ t$	*Präfix-Operatoren*
	$t(t)$	*Applikation*
	$t()$	*Parameterlose Applikation*
	$\texttt{IF}(t,t,t)$	*Einfache Selektion*
	$\texttt{LET}[\text{ldefs}](t)$	*Let-Ausdruck*
$op_{Bin} ::=$		**Binäre infix-Operatoren**
	$\{+,-,*,/,==,!=,\&\&,\|\|\}$	
$op_{Un} ::=$		**Unäre präfix-Operatoren**
	$\{+,-,!\}$	
$ldefs ::=$		**Variablenbindung**
	$v{=}t{,}ldefs$	
	$v{=}t$	
$args ::=$		**Argumentliste**
	$(v)\ args$	
	(v)	*nicht-striktes Argument*
	$(!v)$	*striktes Argument*
$scdef ::=$		**Superkombinatordefinition**
	$v() = t$	
	$v\ args = t$	
$script ::=$		**EML-Skript**
	$scdef$	
	$scdef{,}script$	

Abbildung 8.1: Konkrete Syntax von EML. Operatorpräzedenzen und Assoziativitäten werden von der Grammatik nicht berücksichtigt. Sie ergeben sich aus den Bestimmungen des C++ -Standards.

CoreEML durch Einsatz des speziellen Konstruktors `App` in Applikationen der Ordnung Eins umgewandelt.

8.1.1 Darstellung von CoreEML in C++

Die Darstellung des AST realisieren wir durch Konstruktortypen, also durch Klassen und Klassentemplates, so daß die Struktur eines CoreEML-Programmes sowohl im Typ, als auch im Wert des AST sichtbar wird.

Im wesentlichen halten wir für jede syntaktische Form einen Konstruktor bereit. Einzige Ausnahme sind Formen, die allein der Bildung von Listen dienen (*ldef* und *scr*), die wir auf Polytyplisten abbilden.

Es gibt zwei Besonderheiten: Für C++ -Konstanten führen wir keinen speziellen Konstruktor ein, sondern speichern sie unmittelbar im AST; es exisitiert kein Gegenstück zum Konstruktor `PaSc` in EML. `PaSc` werden wir später zur Behandlung von partieller Applikation einsetzen

Syntax — CoreEML

$t ::=$		**Terme:**
	v	*Funktionale Variable*
	c	C++ -Konstante
	App(t,t)	*Applikation*
	Let$(ldefs, t)r$	*Let-Ausdruck*
$ldefs ::=$		**Variablenbindung**
	sc	
	sc,$ldefs$	
$sc ::=$		**Superkombinatordefinition**
	$t = t$	
$scr ::=$		**CoreEML-Skript**
	sc	
	sc,scr	

Abbildung 8.2: Syntax von CoreEML.

$$
\begin{aligned}
\sigma(\texttt{Var}) &= 1 \\
\sigma(\texttt{Sc}) &= 2 \\
\sigma(\texttt{App}) &= 2 \\
\sigma(\texttt{Let}) &= 2 \\
\sigma(\texttt{PaSc}) &= 2 \\
\sigma(\texttt{Script}) &= 1
\end{aligned}
$$

Abbildung 8.3: Konstruktoren zur Darstellung der abstrakten Syntax von CoreEML.

und im ersten Konstruktorargument den Namen des partiell applizierten Superkombinators und im zweiten Argument, die Liste von Argumenten, die bereits an den Superkombinator übergeben wurden, speichern.

Die Klassen zur Repräsentation der einzelnen Konstruktoren wollen wir nun ein wenig genauer vorstellen.

Beginnen wir mit funktionalen Variablen. Normalerweise sehen wir in der Typdarstellung nur den Typ einer Variablen, so daß wir Variablen nicht wirklich voneinander unterscheiden können. Typ- und Wertdarstellung sind eben nicht wirklich dual. Da wir CoreEML-Programme durch strukturelle Typanalyse in C++ übersetzen wollen, ist es jedoch von essentieller Bedeutung, daß wir Variablen auch innerhalb der Typdarstellung unterscheiden können.

Variablen müssen auf der Typebene unterscheidbar sein. Wir könnten, ähnlich wir bei der Realisierung des *staged interpreters* in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$, Variablen als Instanzen des einstelligen Konstruktors `Var` darstellen. Dann könnten wir nahezu unendlich viele Variablennamen generieren, indem wir z.B. Zahlen (dargestellt als Typterme aus `S` und `N`) als Konstruktorargumente einsetzen.

In C++ können wir auf die aufwendige Zahlendarstellung verzichten, da auch integrale Kon-

Funktionssymbol	Typ	Variablenname
+	Var<-10>	OpPlus
-	Var<-11>	OpMinus
*	Var<-12>	OpMult
/	Var<-13>	OpDivide
==	Var<-14>	OpEqual
!=	Var<-16>	OpNEqual
!	Var<-20>	OpNot
-	Var<-21>	OpUnaryMinus
+	Var<-22>	OpUnaryPlus
IF	Var<-30>	OpIf
LET	Var<-40>	OpLet
jump	Var<-50>	OpJump
call	Var<-60>	OpCall

Abbildung 8.4: Vordefinierte Variablennamen für CoreEML.

stanten als Template-Argumente zulässig sind:

```
template <int NAME>
struct Var {
};
```

Dieses Klassentemplate ist über einen `int`-Wert parametrisierbar, so daß sich eine genügend große Zahl von unterschiedlichen Datentypen generieren läßt.

Instanzen von `Var` mit negativem Wert für `Name`, sind der internen Verwendung vorbehalten und werden zur Darstellung der eingebauten Funktionen (z.B: `+` und `IF`) eingesetzt (siehe Abbildung 8.4).

Bezüglich der Einbettung von EML in C++ ergibt sich, daß funktionale Variablen im C++ -Kontext definiert sein müssen, bevor sie in einem EML-Skript verwendet werden können. Es obliegt dem Programmierer, zu gewährleisten, daß sich Variablen mit verschiedenen Namen auch in ihrem Typ voneinander unterscheiden.

Betrachten wir ein Beispiel:

```
Var<1> fac;
Var<2> x;

( fac(x) = IF( x==0 , 1, x*fac(x-1)) );
```

In EML wird nicht zwischen dem Namen von Superkombinatoren und dem Namen von Parametern unterschieden. Daher müssen sowohl `fac` als auch `x` vor ihrer Verwendung als funktionale Variablen definiert sein. Wichtig ist, daß sie sich im Typ voneinander unterscheiden, da wir sie im Rahmen der strukturellen Typanalyse sonst nicht auseinanderhalten können.

Den Konstruktor `App` bilden wir auf ein Klassentemplate ab, das von zwei Typparametern abhängt: der Funktion `F` und dem Argument `A`:

```
template <typename F,typename A>
struct App {
  App(const F& _f,const A& _a) : f(_f),a(_a) {}
  App(const App<F,A>& rhs) : f(rhs.f), a(rhs.a) {}
  F f;
  A a;
};
```

Die Argumente, die für die Templateparameter `F` und `A` eingesetzt werden, ermöglichen die Darstellung von Funktion und Argument auf der Typebene, während die Objektvariablen `f` und `a` der Darstellung des AST im Speicher dienen.

Werden zum Beispiel die funktionale Variable `f` durch `Var<1>` und die Variable `y` durch `Var<2>` repräsentiert, wird der CoreEML-Term

```
App(App(f,2),y)
```

durch ein Objekt vom Typ

```
App<App<Var<1>,int>,Var<2> >
```

dargestellt.

Das Klassentemplate `App` weicht nur an einer Stelle von der Konstruktor-Klassentemplate Analogie ab, die wir in Abschnitt 5.3.2 aufgestellt haben. In `App` wird ein so genannter Kopierkonstruktor bereitgestellt, mit dessen Hilfe neue Applikationsknoten durch Kopie eines existierenden erstellt werden können. Zwar entspricht dieser Konstruktor exakt dem, was der C++ -Übersetzer auch automatisch generiert hätte, allerdings wirkt die explizite Angabe des Kopierkonstruktors bei vielen C++ -Übersetzern wie ein Schalter, der die Anwendung der *named return value optimization* (NRVO) [106] aktiviert.

Abgesehen von `let`-Ausdrücken stehen jetzt alle Konstruktoren zur Verfügung, um EML-Terme in EMLCore-Terme zu übersetzen. Zum Beispiel wird der EML-Term

$$v_1 + v_2$$

in CoreEML als

$$\texttt{App}(\texttt{App}(\texttt{OpPlus}, v_1), v_2)$$

dargestellt; bzw. als Objekt vom C++ -Typ

```
App<App<OpPlus,Var<1> >,Var<2> >
```

Die infix Operatoren aus EML werden in CoreEML also in Präfix-Notation verwendet. Das gilt auch für das `IF`-Konstrukt aus EML: Aus $\texttt{IF}(t_c,t_{then},t_{else})$ entsteht der CoreEML-Term

$$\texttt{App}(\texttt{App}(\texttt{App}(\texttt{OpIf},t_c),t_{then}),t_{else})$$

Abbildung 8.5 zeigt C++ -Typ und Speicherlayout des zugehörigen Objekts für einen komplexeren CoreEML-Term. Offenbar liegen die Funktionsargumente konsekutiv im Speicher – das werden wir später bei der Implementierung des CoreEML-Interpreters ausnutzen.

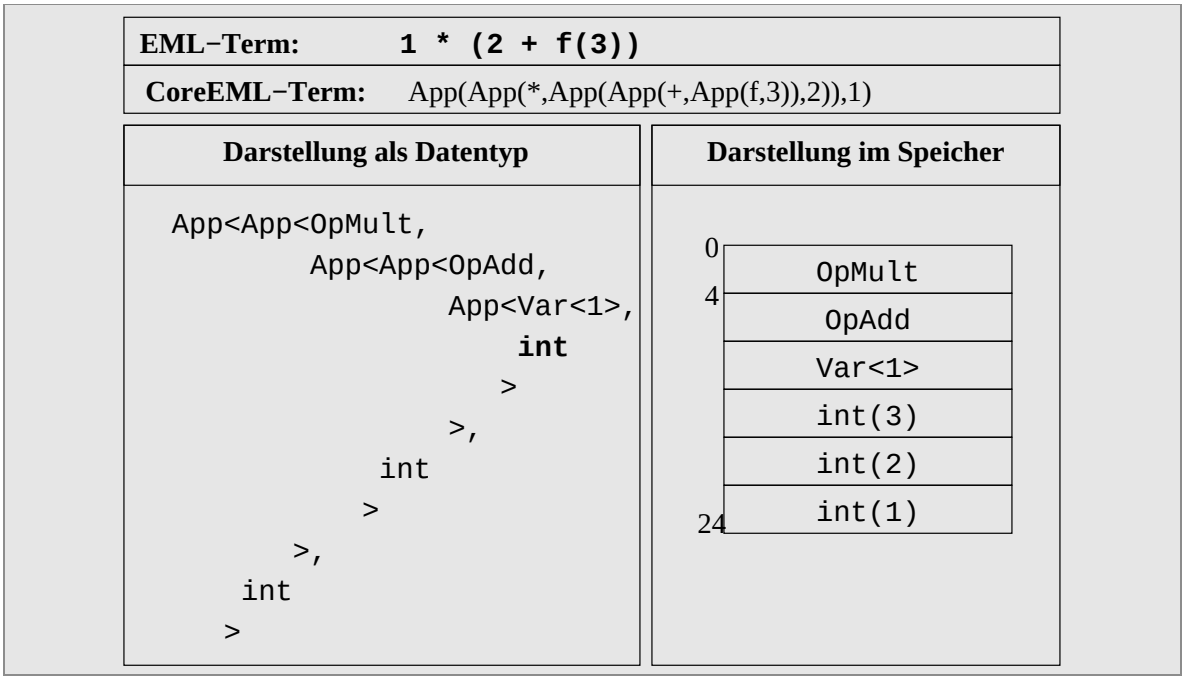


Abbildung 8.5: Verschiedene Darstellungsformen eines EML-Terms.

Eine Superkombinatordefinition besteht aus zwei Komponenten: Einer Liste von Parametern und einem EML-Term. Der Name des Superkombinators ist Bestandteil der Parameterliste – auf die genaue Darstellung dieser Liste gehen wir aber erst später ein.

Superkombinatoren stellen wir als Instanzen des Klassentemplates `Sc` dar:

```
template <typename ARGS,typename T>
struct Sc {
  typedef ARGS Lhs_t;
  typedef T    Rhs_t;
  Sc(const T& _t) : t(_t) {}
  Sc(const Sc& rhs) : t(rhs.t) {}
  T t; // T ist immer von der Form TML::C< .. >
};
```

Im Unterschied zu Applikationsknoten, korrespondieren Typ- und Laufzeitdarstellung hier nicht zu hundert Prozent, da `Sc` kein Feld zur Aufnahme der Argumentliste zur Verfügung stellt. Da EML kein *pattern matching* unterstützt, reicht die Strukturinformation an dieser Stelle aus, um die Parameterliste eindeutig zu beschreiben (unterschiedliche Variablen unterscheiden sich nach Konvention ja auch in ihrem Typ).

Ein `let`-Ausdruck besteht ebenfalls aus zwei Komponenten. In der ersten Komponente wird die Liste der Variablendefinitionen hinterlegt, die mit dem `Let`-Ausdruck eingeführt werden. In der zweiten Komponente wird ein CoreEML-Term hinterlegt.

```
template <typename LDEFS,typename T>
struct Let {
  Let(const T& _t,const LDEFS& _l) : t(_t),l(_l) {}
  Let(const Let& rhs) : t(rhs.t),l(rhs.l) {}
  LDEFS l;
  T     t; // T ist immer von der Form TML::C< .. >
};
```

Ein CoreEML-Skript ist eine Liste von Superkombinatordefinitionen. Zu seiner Darstellung könnten wir unmittelbar auf die übliche Listendarstellung mit `TML::C` und `TML::N` zurückgreifen. Um EMLCore Programme jedoch besser von einfachen Listen abgrenzen zu können, verschachteln wir diese in einer Struktur.

```
template <typename SCLIST>
struct Script {
  Script(const SCLIST& _s) : sclist(_s) {}
  Script(const SCLIST& rhs) : sclist(rhs.sclist) {}
  SCLIST     sclist;
};
```

Der Typ `Script` dient vornehmlich dazu, die Rolle einer Liste durch ein *tag* zu markieren.

Die durch partielle Applikation entstehenden `PaSc`-Knoten speichern wir als Instanzen des folgenden Klassentemplates:

```
template <int NAME,typename STACK>
struct PaSc {
  PaSc(const STACK& s) : theStack(s) {};
  STACK theStack;
};
```

8.2 Parsing von EML-Programmen

Wir hatten bereits gezeigt, daß der C++ -Übersetzer zahlreiche Operatoranwendungen auf Funktions- und Methodenapplikationen abbildet (siehe auch [3, Seite 216]). Abbildung 8.6 vermittelt einen Eindruck, wie sich diese Transformation auf ein einfaches EML-Skript auswirkt.

Unsere Aufgabe besteht nun darin, Funktionen und Operatoren so zu überladen, daß sie – wenn sie auf EML-Codefragmente angewendet werden – ein CoreEML-AST-Fragment als Ergebnis zurückliefern. Diese Technik hatten wir am Beispiel von Interpretern in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ bereits vorgestellt.

Im folgenden werden wir an vielen Stellen Methoden zu den oben definierten Klassentemplates `Var`, `App`, `Sc` und `Script` hinzufügen. Der Übersicht halber werden wir dann nicht die komplette Klassentemplatedefinition erweitern, sondern nur die Definition der Methode außerhalb der Klasse notieren.

```
( fac(x) = If(x==0,1,x*fac(x-1) ),
   square(x) = x * x
)[ fac(square(10)) ];
```

```
operator,( operator=( fac.operator()( x ),
                      If( operator==(x,0),
                          1.
                          operator*(x,
                                    fac.operator()(
                                      operator-(x,1) )
                                   )
                        )
                    ),
           operator=( square.operator()( x ),
                      operator*()(x,x)
                    )
         ).operator[]( fac.operator()( square.operator()(10 ) ) );
```

Abbildung 8.6: Ein EML-Skript (oben) aus dem Blickwinkel des C++ -Übersetzers (unten).

8.2.1 Funktionsapplikation

Betrachten wir die Funktionsapplikation $t_1(t_2)$ und vergessen für den Moment, daß es sich um einen EML-Ausdruck handelt. Der Ausdruck $t_1(t_2)$ ist genau dann ein gültiger C++ -Ausdruck, wenn die Variable t_1 mit der Syntax zum Aufruf von Funktionen kompatibel ist. Das ist der Fall, wenn t_1 ein Zeiger auf eine unäre Funktion, oder aber ein Objekt ist, in dem die Methode `operator()` geeignet überladen wurde.

Im einfachsten Fall handelt es sich bei t_1 um eine funktionale Variable, wir erweitern die Templateklasse `Var` daher um einen Klammeroperator:

```
template <int NAME>
template <typename A>
inline App<Var<NAME>,A> Var<NAME>::operator()(const A& a) const {
  return App<Var<NAME>,A>(*this,a);
}
```

Anmerkung 8.1. Die doppelte `template`-Zeile ist kein Druckfehler, sondern erforderlich. Wir definieren hier eine Methode für das Klassentemplate `Var<NAME>` (erste `template`-Zeile). Diese Methode ist selbst ein Template, welches vom Typparameter `A` abhängt (zweite `template`-Zeile). ◊

Wird eine funktionale Variable `v` vom Typ `Var<1>` auf ein Argument `a` vom Typ `A` angewendet (`f(a)`), so interpretiert der C++ -Übersetzer dies als den Aufruf der Methode `operator()` der Klasse `Var<1>`:

```
v.operator()(a)
```

Das Ergebnis dieser Berechnung ist ein Objekt vom Typ `App<Var<1>,A>`; entsprechend dem CoreEML-Ausdruck `App(f,a)`.

Da `operator()` mit Argumenten beliebigen Typs kompatibel ist, können wir bereits jetzt komplexere EMLCore-Objekte parsen, z.B. wird der Term `f(f(x))` korrekt erkannt und in `App(f, App(f, x))` übersetzt. Wir können funktionale Variable aber auch auf beliebige C++ -Ausdrücke anwenden. Sei `d` eine C++ -Variable vom Typ `double` und `sin` die Sinus-Funktion aus der C++ -Mathematikbibliothek, dann wird der Term `f(d=sin(3.9))` als korrekter EML-Term erkannt – der Term `d=sin(3.9)` fällt in die Menge der C++ -Konstanten.

Die parameterlose Applikation realisieren wir nach demselben Prinzip durch Hinzufügen einer parameterlosen Methode `operator()` zum Klassentemplate `Var`. Aus Konsistenzgründen generieren wir auch hier ein Objekt vom Typ `App`. Der Argumenttyp / das Argument sind nicht weiter von Bedeutung – wir greifen auf das Symbol zur Darstellung der leeren Liste zurück:

```
template <int NAME>
inline App<Var<NAME>,TML::N> Var<NAME>::operator()() const {
  return App<Var<NAME>,TML::N>(*this,TML::N());
}
```

Wurde eine Funktionsvariable appliziert, so entsteht ein Applikationsobjekt. Dieses stellt selbst wieder eine Funktion dar und kann auf weitere Argumente angewendet werden, wodurch ein neues Applikationsobjekt entsteht. Wir überladen daher den Klammeroperator des Klassentemplates `App`:

```
template <typename F,typename A>
template <typename A2>
App<App<F,A>,A2 > App<F,A>::operator()(const A2& a2) const {
  return App<App<F,A>,A2>( *this, a2 );
}
```

Seien nun `f` vom Typ `Var<1>`, `x` von einem beliebigen Typ `A` und `y` ebenfalls von einem beliebigen Typ `A2`. Der Ausdruck `f(x)(y)` wird vom C++ -Übersetzer so behandelt, als hätte der Programmierer

```
(f.operator()(x)).operator()(y)
```

geschrieben. Aufgrund der strikten Auswertung von C++ wird zunächst der Ausdruck

```
f.operator()(x)
```

abgearbeitet, was zu einem Objekt vom Typ `App<Var<1>,A >` führt. Dies ist ein Klassentyp, der vom Klassentemplate `App` abgeleitet ist, welches wir soeben mit einer `operator()`-Methode ausgestattet haben. Aus Sicht des C++ -Übersetzers ist der Ausdruck `f(x)(y)` damit äquivalent zu:

```
(App<Var<1>,A>(f,x)).operator()(y)
```

Der erste Teil des Ausdrucks (`App<Var<1>,A>(f,x)`) generiert ein temporäres Objekt vom Typ `App<Var<1>,A>`. Anwendung der Methode `operator()` aus dem `App` Klassentemplate führt zu einem temporären Objekt, dessen Typ die syntaktische Struktur des EML-Terms widerspiegelt:

```
App<App<Var<1>,A>,A2>(  (App<Var<1>,A>(f,x))( f,x ), y )
```

Obwohl von der EML-Syntax vorgesehen, schließen wir aus, daß C++ -Konstanten (unter Berücksichtigung der partiellen Auswertung also allgemeine C++ -Ausdrücke, die keine funktionalen Variablen enthalten) die Rolle von EML-Funktionen übernehmen können.

8.2.2 Infix- und präfix-Operatoren

Die infix-Schreibweise von binären Operatoren in EML-Termen gewährleisten wir durch entsprechendes Überladen der binären Operatoren in C++ . Operatorpräzedenzen und Assoziativitäten richten sich in EML daher nach den Regeln, wie sie im C++ -Standard vereinbart wurden.

Wie zuvor auch, werden Objekte generiert, deren Typ Element der Sprache CoreEML ist. Hierbei müssen wir sehr sorgfältig vorgehen. Das Überladen der Operatoren für beliebige Argumenttypen, so wie wir es bei `operartor()` getan haben, kommt nicht in Frage. Würden wir beide Funktionsargumente als beliebig vereinbaren, ergeben sich bei der Auflösung von Überladung Doppeldeutigkeiten für jeden weiteren definierten Operator.

Wir beschränken uns daher auf die Fälle, bei denen funktionale Variablen involviert sind. Alle anderen Fälle überlassen wir dem „partiellen Auswerter". Abbildung 8.7 zeigt die zum Parsing notwendigen Überladungen von `operator+`. Alle anderen binären und unären Operatoren werden analog behandelt (für unäre Operatoren sind natürlich nur zwei Funktionstemplates erforderlich).

8.2.3 Einfache Selektion

Leider kann `operator?` in C++ nicht überladen werden, weshalb wir für EML das Konstrukt `If` einführen.

Zum Parsing der einfachen Selektion stellen wir drei Funktionstemplates zur Verfügung:

```
template <typename C>
App<OpIf,C> IF(const C& c) {
  return App<OpIf,C>(OpIf(),c);
}

template <typename C,typename T>
App<App<OpIf>,C>,T> IF(const C& c,const T& t) {
  return App<App<OpIf,C>,T>( App<OpIf,C>(OpIf(),c), t);
}

template <typename C,typename T,typename E>
App<App<App<OpIf,C>,T>,E> IF(const C& c,const T& t,const E& e) {
  return App<App<App<OpIf,C>,T>,E>( App<App<OpIf,C>,T>( App<OpIf,C>(OpIf(),c),
                                                        t
                                                      ),
                                    e
                                  );
}
```

```
// Var + Var
template <int A,int B>
App<App<OpPlus,Var<A> >,Var<B> > operator+(cosnst Var<A>& a,const Var<B>&
b) {
  return App<App<OpPlus,Var<A> >,Var<B> >(
              App<OpPlus,Var<A> >(OpPlus(), A ), B);
}
// Var + T
template <int A,typename T>
App<App<OpPlus,Var<A> >,T> operator+(const Var<A>& a,const T& t) {
  return App<App<OpPlus,Var<A> >,T>( App<OpPlus,Var<A> >(OpPlus(),a), t );
}
// T + Var
template <typename T,int A>
App<App<OpPlus,T >,Var<A> > operator+(const T& t,const Var<A>& a) {
  return App<App<OpPlus,T >,Var<A> >( App<OpPlus,T >(OpPlus(),t), a );
}
// App + T
template <typename F,typename A,typename T>
App<App<OpPlus, App<F,A> >,T> operator+(const App<F,A>& app,const T& t) {
  return App<App<OpPlus, App<F,A> >,T>( App<OpPlus,App<F,A> >(OpPlus(),app)
, t);
}
// T + App
template <typename T,typename F,typename A>
App<App<OpPlus,T>, App<F,A> > operator+(const T& t,const App<F,A>& app) {
  return App<App<OpPlus,T>, App<F,A> >( App<OpPlus,T>( OpPlus(),t ), app );
}
```

Abbildung 8.7: Parsing von binären infix-Operatoren am Beispiel der Addition.

8.2.4 Superkombinatordefinitionen und Skripte

Eine Superkombinatordefinition ist eine Zuweisung, bei der sich auf der linken Seite eine EML-Variable, oder ein EML-Term befindet, während die rechte Seite ein EML-Term ist.

Um das Parsing von Superkombinatordefinitionen zu ermöglichen, müssen wir für jedes Objekt, welches auf der linken Seite der Zuweisung auftauchen kann, den Zuweisungsoperator überladen. Ist `o` ein Objekt von einem Klassentyp mit überladenem Zuweisungsoperator wird der Programmcode

```
o = e;
```

vom C++ -Übersetzer so behandelt, als hätte man `o.operator=(e)` geschrieben.

Die Klassentemplates `Var` und `App` werden daher um entsprechende Zuweisungsoperatoren ergänzt (wir zeigen nur den Fall für das Template `App`):

```
// Sc, Sc
template <typename LHS,typename RHS, typename LHS2,typename RHS2>
Script< TML::C<Sc<LHS,RHS>, TML::C<Sc<LHS2,RHS2>, TML::N > > > operator,
  (const Sc<LHS,RHS>& sc1, const Sc<LHS2,RHS2>& sc2) {
    return Script< TML::C<Sc<LHS,RHS>, TML::C<Sc<LHS2,RHS2>, TML::N > > >(
             TML::cons(sc1,TML::cons(sc2))
           );
}
// Script, Sc
template <typename SLIST,typename LHS,typename RHS>
Script< TML::C<Sc<LHS,RHS>, SLIST> > operator,(const Script<SLIST>& script,
                                               const Sc<LHS,RHS>& sc) {
  return Script<TML::C<LHS,RHS> >( cons(sc,script.sclist) );
}
// Sc, Script
template <typename LHS,typename RHS,typename SLIST>
Script< TML::C<Sc<LHS,RHS>, SLIST> > operator,(const Sc<LHS,RHS>& sc,
                                               const Script<SLIST>& script) {
  return Script<TML::C<LHS,RHS> >( cons(sc,script.sclist) );
}
```

Abbildung 8.8: Parsing von EML-Skripten.

```
template <typename F,typename A>
template <typename T>
Sc< App<F,A>, T> App<F,A>::operator=(const T& t) {
  return Sc<App<F,A>,T>( *this, t );
}
```

Ein EML-Skript ist schließlich eine Sequenz von Superkombinatordefinitionen, die durch Kommata voneinander zu trennen sind. Listen stellen wir als Polytypliste, also unter Rückgriff auf die bereits in Abschnitt 5.3.2 (Seite 123) vorgestellten Konstruktoren `C` und `N` dar. Kurze Bezeichnernamen global zu definieren ist allerdings unglücklich, da die Gefahr, daß es zu Doppeldeutigkeiten kommt, hoch ist. Wir gehen daher von nun an davon aus, daß die Listenkonstruktoren `C` und `N`, sowie sämtliche Listenoperationen im Namensraum `TML` vereinbart wurden. Es fehlt also nur noch eine geeignete Überladung von `operator,`. Leider kommen wir mit einer Überladung nicht aus, da wir EML-Programme in den Datentyp `Script` einschließen wollen (siehe Abbildung 8.8).

8.2.5 `let`-Ausdrücke

Das Parsing von *let*-Ausdrücken ist etwas komplexer, da es hier nicht ausreicht, einen Operator oder eine Funktion zu überladen. Hinter dem EML-Schlüsselwort `LET` steht, eingeschlossen von eckigen Klammern eine Liste von *let*-Bindings. Direkt im Anschluß folgt ein EML-Term, der in runde Klammern zu setzen ist.

Das eckige Klammerpaar `[` und `]`ist in C++ ein unärer postfix-Operator, auch *subscript operator* genannt. Wie der Zuweisungs- und Klammeroperator auch, kann er für Objekte

```
struct LetSymbol {
private:
  // Templateklasse zur Repräsentation des
  // Zwischenergebnisses.
  template <typename LETLIST>
  struct LetHelp {
    LetHelp(const LETLIST& l) : list(l) {}
    template <typename T>
    inline App<OpLet,Let<LETLIST,T> > operator()(const T& t) const {
      return App<OpLet,Let<LETLIST,T> >( OpLet(), Let<LETLIST,T>( list, t ) );
    }
    LETLIST list;
  };
public:
  // Subscript-Operatoren
  // a) Nur eine Variable wird gebunden
  template <typename LHS,typename RHS>
  inline LetHelp<TML::C<Sc<LHS,RHS>,TML::N> > operator[](const Sc<LHS,RHS>&
l) const {
    return LetHelp<TML::C<Sc<LHS,RHS>,TML::N> >( cons(l,TML::N()) );
  }
  // b) Mehrere Let-Bindings
  template <typename SCLIST>
  inline LetHelp<SCLIST> operator[](const Script<SCLIST>& prg) const {
    return LetHelp<SCLIST>( p.getProgram() );
  }
} LET;
```

Abbildung 8.9: Parsing von `let`-Ausdrücken. Das EML-Schlüsselwort `Let` ist in Wirklichkeit ein Objekt vom Typ `LetSymbol`.

überladen werden. Das EML-Schlüsselwort `LET` stellen wir daher als ein Objekt vom Typ `LetSymbol` dar. Ein Ausdruck der Form

```
LET[ f(x)=x+2 ]( f(2) )
```

wird vom C++ -Übersetzer genauso behandelt wie der Ausdruck

```
(LET.operator[]( f(x)=x+2 )).operator()( f(2) )
```

Zur Gewährleistung der *let*-Syntax muß sich das Ergebnis des Indizieroperators wie eine Funktion verhalten, die auf EML-Terme angewendet werden kann. Am einfachsten erreichen wir dieses Verhalten, indem wir ein neues, internes Funktionssymbol `OpLet` vereinbaren und dies zusammen mit dem eigentlichen `Let`-Knoten in einen `App`-Knoten packen:

$$\texttt{App}(\texttt{OpLet}, \texttt{Let}(f(x) = 2, f(2)))$$

Das Parsing läuft zweistufig ab. Der Indizieroperator der Klasse `LetSymbol` gibt als Zwischenprodukt ein Objekt vom Typ `LetHelp<SCLIST>` zurück, wobei `SCLIST` dem Typ der

Let-Bindings, entspricht, die dem Indizieroperator übergeben wurden. Hierbei handelt es sich entweder um eine einfache Superkombinatordefinition, oder aber um mehrere, die durch Kommata voneinander zu trennen sind. Diese werden von unserem Parser als Skripte erkannt (siehe zweite Überladung von `operator[]` in Abbildung 8.9).

Streng genommen, müßten wir noch prüfen, ob es sich wirklich um *let*-Bindings handelt, d.h. auf den linken Seiten jeweils nur eine Variable steht. Ein entsprechendes Metaprogramm ist leicht zu erstellen und wir wollen auf seine Angabe aus Platzgründen verzichten.

Im Zwischenobjekt werden die Let-Bindings gespeichert und der im Klassentemplate `LetHelp` definierte Funktionsemulationsoperator ermöglicht, wie gewünscht, daß wir ein `LetHelp`-Objekt auf einen beliebigen Ausdruck anwenden können.

8.2.6 EML-Programme

Damit können wir vollständige EML-Skripte in unsere interne Darstellung überführen. Als Ergebnis des Parsing erhalten wir entweder ein Objekt vom Typ `Script<T>`, oder ein Objekt vom Typ `Sc<LHS,RHS>`, wobei `T`, `RHS` und `LHS` Platzhalter für beliebige CoreEML-Terme sind.

Aus einem EML-Skript wird ein Programm, indem wir den Indizieroperator auf einen EML-Term anwenden. Wir statten die Klassentemplates `Sc` und `Script` mit einem passenden Operator aus, den wir hier allerdings nur skizzieren können, da uns noch Grundlagen bezüglich der Typinferenz und der Code-Generation fehlen:

```
template <typename SCLIST>
struct Script {
  typedef SCLIST TheScript;
  template <typename EMLEXPRESSION>
  // Aufruf des Metaprogrammes inferType zur Berechnung
  // des Ergebnistyps
  typename inferType<TheScript,TML::N,TML::N,EMLEXPRESSION>::RET
    operator()(const EMLEXPRESSION& e) const {
    // Übersetze das EML-Skript durch Aufruf der Metafunktion
    // translateCoreEML.
    (translateCoreEML<TheScript,EMLEXPRESSION>::run)(s,e);
  }
  SCLIST s;
};
```

In der Implementierung des Indizieroperators tauchen zwei Metaprogramme auf. Das Metaprogramm `inferType` berechnet aus dem EML-Skript und dem auszuwertenden Term den Typ des Ergebniswertes. Die Bedeutung der beiden anderen Argumente (`TML::N`) erklären wir später.

Das Metaprogramm `translateCoreEML` ist ein Code-Generator, der aus der Typdarstellung von Skript und Term Programmcode zur Auswertung des CoreEML-Programmes generiert. Der dabei zurückgelieferte Wert muß natürlich mit dem durch `inferType` berechneten kompatibel sein.

In den folgenden zwei Kapiteln werden wir die Hintergründe und die Implementierung dieser beiden Metafunktionen genauer beschreiben.

Kapitel 9

Typsystem und Typinferenz für EML

In vielen funktionalen Programmiersprachen kann man auf Typannotationen verzichten. Ein Typinferenzverfahren versucht einer Funktionen einen Typ zuzuordnen, der möglichst alle Werte beschreibt, mit denen sie fehlerfrei ausgeführt werden kann – einen solchen Typ nennt man auch einen **prinzipalen Typ**.

Die Zuordnung eines Typs zu einer beschränkt polymorphen Funktion setzt voraus, daß man Einschränkungen an einen Typparameter explizit in der Typsprache ausdrücken kann (z.B. Typklassen in Haskell: Im Typ `Eq a => a -> a` wird die Typvariable `a` auf Typen eingeschränkt, die der Typklasse `Eq` angehören).

Die Typsprache von C++ stellt diese Möglichkeit leider nicht zur Verfügung. Funktionsbezeichner können nahezu beliebig überladen werden, so daß sich in C++ kein prinzipaler Typ für ein Funktionssymbol finden läßt. Der allgemeinste Typ $\forall\alpha :: *.\forall\beta :: *.\alpha \to \beta$ ist natürlich kein brauchbarer Kandidat, da er kein prinzipaler Typ ist.

Da wir die eingebauten Operatoren von EML auf C++ abbilden, werden wir folglich keinen allgemeinsten Typ für EML-Superkombinatoren angeben können. Das ist insofern nicht weiter tragisch, als daß wir EML-Skripte weder separat übersetzen wollen, noch ein Modul-System für EML realisieren möchten. Uns reicht ein Verfahren, welches uns zu einem EML-Programm berechnet, welchen Typ das Ergebnis haben wird.

Unser Lösungsansatz beruht auf der Idee der **abstrakten Interpretation** [125, Kapitel 4]. Anstatt ein EML-Programm auf „normalen" Werten auszuführen, rechnen wir auf der abstrakten *domain* der Typen. Konstanten und Variablen werden durch ihren Typ, Grundfunktionen durch die ihnen zugeordnete Typfunktion ersetzt.

Das Programm

```
( f(x)(y) = x + y )[ f(1)(2) ]
```

erscheint unter der abstrakten Interpretation folglich als:

$$(\mathtt{f}_{\mathbb{T}}\ \mathtt{x}\ \mathtt{y} = +_{\mathbb{T}}\ \mathtt{x}\ \mathtt{y})[\mathtt{f}_{\mathbb{T}}[\mathtt{int}][\mathtt{int}]]$$

Dabei bezeichnet $+_{\mathbb{T}}$ die abstrakte Interpretation der Additionsfunktion, also die zu `operator+` gehörende Typfunktion.

Da die eingebaute Additionsfunktion zweier integer-Zahlen ein Ergebnis vom Typ `int` liefert, kann der Term $\mathtt{f}_{\mathbb{T}}[\mathtt{int}][\mathtt{int}]$ mit der Funktionsgleichung zu $\mathtt{f}_{\mathbb{T}}$ zum Typ `int` reduziert werden.

Leider gewährt C++ keinen direkten Zugriff auf die zu einer Funktion oder einem Operator gehörende Typfunktion. Wir müssen diese daher als Typfunktion (Metafunktion) nachimplementieren. Um auch auf Objekten rechnen zu können, muß dabei die Klassenhierarchie berücksichtigt werden. Mehrfachvererbung wird sich insbesondere bei der Berechnung des Ergebnistyps von rekursiven Funktionen als problematisch erweisen.

9.1 Typfunktion für Grundfunktionen

Wenn wir die zu einer C++ -Funktion gehörende Typfunktion dem C++ -Standard nachempfinden wollen, müssen wir Typ-Konvertierungen, Typ-Promotionen und überladene Funktionen berücksichtigen.

Selbst dann, wenn man sich auf Grundfunktionen und Basistypen beschränkt, sind die Vorgaben des C++ -Standards sehr aufwendig, so daß wir einige Vereinfachungen vornehmen.

Zu den Basistypen (siehe [3, Seite 53 ff]), die wie in EML berücksichtigen, gehören

- **Integrale Typen:** `char`, `short int`, `int`, `long int`; jeweils mit und ohne Vorzeichen (`signed`/`unsigned`)
- **Fließkomma-Typen:** `float`, `double` und `long double`
- **Boolescher Typ:** `bool`

Zeiger- und Referenztypen lassen wir zunächst außen vor.

Grundfunktionen, die wir in EML zur Verfügung stellen, sind

- **Arithmetische Operatoren:** `operator+`, `operator*`, `operator/` und `operator-`
- **Vergleichsoperatoren:** `operator==` und `operator!=`
- **Negationsoperator:** `operator!`
- **Konditionaloperator:** `operator?:`

Arithmetische Operationen sind mit allen Basistypen verträglich – einschließlich dem Typ `bool`. Zum Beispiel ist der Term `true + true` gültig, da boolesche Werte automatisch in Werte vom Typ `int` konvertiert werden können, wobei `true` dem Wert `1` und `false` dem Wert `0` zugeordnet ist.

Abgesehen vom Typ `bool`, liefern alle arithmetischen Operatoren bei typgleichen Argumenten ein Ergebnis, daß dem Typ der beiden Argumente gleicht. Komplizierter wird es, wenn die Argumente unterschiedliche Typen haben. Dann wird als Ergebnistyp der konzeptionell größere der beiden Argumenttypen gewählt.

Mit „konzeptionell größer" ist gemeint, daß die zum kleineren Typ gehörende Wertemenge in der zum größeren Typ gehörenden enthalten ist. Einzige Ausnahme bilden die kleinen integer-Typen `short int` und `char`: Hier wird grundsätzlich ein Ergebnis vom Typ `int` (mit passendem Vorzeichen) generiert. Der Grund für diese Sonderbehandlung liegt darin, daß der C++ -Standard für den Typ `int` vorschreibt, daß dessen Werte in ein Maschinenregister passen müssen – Operationen auf `int`-Werten können daher sehr schnell abgearbeitet werden.

Vergleichsoperatoren und Negationsoperator sind ebenfalls mit allen Grundtypen verträglich und liefern immer einen Wert vom Typ `bool` als Ergebnis. Daß der Negationsoperator auch auf nicht-boolesche Werte anwendbar ist, ist eine Konsequenz der in C++ eingebauten impliziten Konvertierungen. Mit der *boolean conversion* (siehe [3, Seite 61]) können Werte mit integralem Typ in boolesche Werte konvertiert werden. Die *floating point integral conversion* (siehe [3, Seite 60]) erlaubt die Konvertierung eines Fließkomma-Wertes in einen Wert mit integralem Typ. Wird der Negationsoperator auf einen Fließkomma-Wert angewendet, so wird dieser (automatisch) in einen integralen Typ und anschließend in einen booleschen Typ konvertiert.

In C++ unterscheidet man Typ-Promotionen und Typ-Konvertierungen. Bei einer Typ-Promotion wird ein Wert an eine Variable mit konzeptionell größerem Typ gebunden. Es ist garantiert, daß dabei keine Information verloren geht. Im Gegensatz dazu ist eine Konvertierung eine semantische Operation, die einen Wert ändert. Zum Beispiel werden bei der *floating point integral conversion* die Nachkommastellen abgeschnitten und sollte sich der verbleibende Wert nicht als integeraler Wert darstellen lassen, ist das Verhalten laut C++ -Standard sogar undefiniert.

Für EML wollen wir daher nur Typ-Promotionen erlauben und der Negationsoperator soll nur auf boolesche Werte anwendbar sein. Es ist klar, daß wir diese Einschränkungen nur für Terme erwirken können, die funktionale Variablen enthalten. Aus Sicht des Interpreters ist der Ausdruck `!1.0` vom Typ `bool`, also eine Konstante. Wir können lediglich verhindern, daß z.B. die funktionale Variable `v` im Term `!v` durch einen Wert ersetzt wird, dessen Typ nicht vom Typ `bool` ist.

Typ-Promotionen implizieren eine Ordnung über den Typen. Diese Ordnung nennen wir **Untertyprelation** und bezeichnen sie mit $<:$. Gilt $T_1 <: T_2$, so sind Werte vom Typ T_1 überall dort einsetzbar, wo ein Wert vom Typ T_2 erwartet wird – $<:$ reflektiert die Inklusionsbeziehung der Interpretation eines Typs.

Der C++ -Standard sortiert aber nicht nur Grundtypen, sondern auch Klassen. Eine Ordnung über Klassen läßt sich unmittelbar aus dem Quelltext ablesen: ist z.B. eine Klasse `D` von einer Klasse `B` abgeleitet, können sich Objekte vom Typ `D` wie Objekte vom Typ `B` verhalten. Es gilt also $\texttt{D} <: \texttt{B}$.

Wir wollen die Untertyprelation genauer fassen:

Definition 9.1 (Die Untertyprelation $<:$)**.** Sei $\mathcal{C}_\mathbb{T}(n)$ die Menge der in der n-ten Zeile eines C++ -Programmes sichtbaren Typsymbole. Dann ist $<: : \mathcal{C}_\mathbb{T}(n) \times \mathcal{C}_\mathbb{T}(n)$ definiert durch:

1. $\forall T \in \mathcal{C}_\mathbb{T} : T <: T$
2. `signed char` $<:$ `short int` $<:$ `int` $<:$ `long int` $<:$ `bool`
3. `unsinged char` $<:$ `unsigned short int` $<:$ `unsigned int` $<:$ `unsigned long int` $<:$ `bool`
4. `float` $<:$ `double` $<:$ `long double`
5. Für jede Klassendeklaration `class C :` $\texttt{S}_1, \ldots, \texttt{S}_n$ im Sichtbarkeitsbereich von Zeile n gilt weiter:
   - $\texttt{C} <: \texttt{S}_1, \cdots, \texttt{C} <: \texttt{S}_N$
   - $\texttt{S}_1 <: D \Rightarrow \texttt{C} <: D$ mit $D \in \mathcal{C}_\mathbb{T}$.

```
max := Λα :: ∗.Λβ :: ∗.if α <: β then β else
        if β <: α then α else FAIL

arithBinReturn := Λα :: ∗.
  match α with γ × δ ⇒
    if (isBaseType[γ]) and (isBaseType[δ]) then max[γ][δ]
  else FAIL

boolBinReturn := Λα :: ∗.
  match α with γ × δ ⇒
    if (isBaseType[γ]) and (isBaseType[δ]) then bool
  else FAIL

bop_T := Λα :: ∗.
  match α with
    OpPlus × γ × δ ⇒    arithBinReturn[γ × δ]
    OpSub × γ × δ ⇒     arithBinReturn[γ × δ]
    OpDivide × γ × δ ⇒  arithBinReturn[γ × δ]
    OpMult × γ × δ ⇒    arithBinReturn[γ × δ]
    OpEq × γ × δ ⇒      boolBinReturn[γ × δ]
    OpNEq × γ × δ ⇒     boolBinReturn[γ × δ]
  else FAIL

uop_T := Λα :: ∗.
  match α with
    OpUnaraPlus × γ ⇒  γ
    OpUnaryMinus × γ ⇒ γ
    OpNot × bool ⇒      bool
    OpNot × int ⇒       bool
  else FAIL
```

Abbildung 9.1: Vereinfachte Typfunktion für unäre und binäre Operatoren, wie sie zur Typberechnung von EML eingesetzt werden.

Die Relation <: läßt sich natürlich leicht durch eine Typfunktion beschreiben, die `TRUE` zurückliefert, wenn die zu vergleichenden Typen in <: enthalten sind, und `FALSE`, falls dies nicht der Fall ist.

Zur Implementierung der Typfunktionen für Grundoperationen greifen wir auf die Untertyprelation zurück und verzichten auf die Sonderbehandlung der kleinen integralen Typen.

Um nicht für jeden Operator eine eigene Typfunktion erstellen zu müssen, übergeben wir den Typfunktionen neben den Argumenttyp(en) auch das Operatorsymbol und unterscheiden nur zwischen unären und binären Operatoren.

Die Funktion $\texttt{bop}_{\mathbb{T}}$ berechnet den Ergebnistyp von binären Operatoren. Sie übernimmt ein Tripel aus Operatorname (den wir als Instanz des Konstruktors `Var` darstellen) und den beiden Argumenttypen. Analog erwartet die Funktion $\texttt{uop}_{\mathbb{T}}$, die den Ergebnistyp von unären Operationen berechnet, ein Tupel aus Operatorname und Argumenttyp als Argument (siehe Abbildung 9.1).

EMLs `If`-Anweisung bilden wir auf den Fragezeichenoperator ab, der sich in C++ als dreistellige Funktion `operator?:` darstellt. Allerdings ist die vom C++ -Standard vorgesehene

Typfunktion für diesen Operator ein wenig unflexibel, was die Behandlung von Objekten angeht. Vereinfacht dargestellt, ist diese wie folgt definiert:

- Kann das erste Argument an `operator?:` nicht in einen Wert vom Typ `bool` konvertiert werden, liegt ein Typfehler vor und die Typfunktion liefert `FAIL` als Ergebnis.
- Sind zweites und drittes Argument von einem Grundtyp, wird als Ergebnis der konzeptionell größere der beiden Typen gewählt (dieser läßt sich zum Beispiel durch Anwendung von $\texttt{bop}_{\mathbb{T}}$ ermitteln).
- Führt das zweite Argument zu einem Klassentyp $\texttt{T}_1$ und das dritte zu einem Klassentyp $\texttt{T}_2$, so wird getestet, ob Werte vom Typ $\texttt{T}_1$ in Werte vom Typ $\texttt{T}_2$ konvertiert werden können und umgekehrt. Sind Konvertierungen in beide Richtungen möglich, so liegt ein Typfehler vor; ansonsten ist das Ergebnis von dem Typ, in den der andere konvertiert werden konnte.

Im objektorientierten Kontext kann das Verhalten von `opertator?:` unnatürlich erscheinen, da seine Typfunktion die Klassenhierarchie außer acht läßt. Betrachten wir dazu folgendes Codefragment:

```
class Base {} b;
class D1 : public Base {} d1;
class D2 : public Base {} d2;
// Typfehler !
(1==2) ? d1 : d2;
```

Es existiert keine Konvertierungsvorschrift, um aus einem Wert vom Typ `D1` einen Wert vom Typ `D2` zu machen, oder umgekehrt – es liegt ein Typfehler vor und die zu `operator?:` gehörende Typfunktion würde `FAIL` als Ergebnis liefern.

$$
\begin{array}{l}
\texttt{If}_{\mathbb{T}} := \Lambda\alpha. \\
\quad \texttt{match } \alpha \texttt{ with } \beta \times \gamma \times \delta \Rightarrow \\
\quad\quad \texttt{If } \beta = \texttt{bool then} \\
\quad\quad\quad \texttt{If } \exists \bigsqcup\{\gamma, \delta\} \texttt{ then } \bigsqcup\{\gamma, \delta\} \\
\quad\quad\quad \texttt{else FAIL} \\
\quad\quad \texttt{else FAIL} \\
\quad \texttt{else FAIL}
\end{array}
$$

Abbildung 9.2: Typfunktion zu `If`. Offenbar ist $\texttt{If}_{\mathbb{T}}$ eine strikte Funktion.

Im Sinne der objektorientierten Programmierung erscheint es natürlicher, nach der kleinsten gemeinsamen Superklasse von `D1` und `D2` suchen. Ist diese eindeutig bestimmbar, was wegen möglicher Mehrfachvererbung nicht immer der Fall sein muss, verwendet man diese Superklasse als Ergebnistyp von `operator?:`. Anders ausgedrückt untersucht man, ob die Menge $\{\texttt{D1}, \texttt{D2}\}$ ein Supremum bezüglich der Relation $<:$ hat. Existiert dies, so bezeichnet es den Ergebnistyp von `operator?:`, ansonsten liegt ein Typfehler vor und wir geben $\bot$ als Ergebnis zurück.

Diese Strategie hat den erfreulichen Nebeneffekt, daß die Differenzierung zwischen Klassen- und Grundtypen entfallen kann, was die Typfunktion zu `operator?:` erheblich vereinfacht (siehe Abbildung 9.2).

```
struct Vec {
  Vec(int _x1,int _x2,int _x3) : x1(_x1),x2(_x2),x3(_x3) {}
  int x1,x2,x3;
};

struct colorVec : public Vec {
  enum color {white,black};
  colorVec(Vec v,color _c=black) : Vec(v), c(_c) {};
  color c;
};

Vec operator+(const Vec& a,const Vec& b) {
  return Vec(a.x1+b.x1, a.x2+b.x2, a.x3+b.x3);
};
```

Abbildung 9.3: Operatorüberladung und Subtyping. `operator+` kann auch auf Objekte vom Typ `colorVec` angewendet werden. Der Ergebnisvektor hat dann die Farbe schwarz.

Sollte der Benutzer Operatoren überladen, so kann er die Typfunktion dank der Erweiterbarkeit von Metaprogrammen sehr leicht anpassen.

Dabei ist jedoch Vorsicht geboten, wenn man Subtyping korrekt berücksichtigen will: `Match` (und entsprechend das *pattern matching* in C++) lassen Subtyping nämlich unberücksichtigt.

Betrachten wir hierzu den Beispielcode aus Abbildung 9.3. `operator+` wurde überladen und kann zusätzlich mit folgenden Typkombinationen aufgerufen werden: `Vec`×`Vec`, `Vec`×`colorVec`, `colorVec`×`Vec` und `colorVec`×`colorVec`. Um diese Kombinationen auch im EML-Kontext zuzulassen, muß die Typfunktion $bop_{\mathbb{T}}$ um entsprechende Behandlungsfälle erweitert werden. Würden wir in `Match` Subtyping berücksichtigen, käme man mit einem zusätzlichen Behandlungsfall aus.

Im nächsten Abschnitt gehen wir der Frage nach, wie man den Ergebnistyp einer Superkombinatorapplikation bestimmt.

9.2 Superkombinatoren und Typfunktionen

Superkombinatoren entstehen durch Komposition von Grundfunktionen. Wenn wir ein EML-Programm unter der abstrakten Interpretation ausführen, ersetzen wir Grundfunktionen durch ihre zugehörige Typfunktion und Konstanten und C++ -Variablen durch ihr Typsymbol.

Wie wir Eingangs an einem Beispiel gezeigt haben, kann man sehr leicht von einem EML-Skript zu einer Menge von Funktionsgleichungen übergehen, die der abstrakten Interpretation entspricht. Dem Skript aus Abbildung 9.4 würden wir bei einfacher Vorgehensweise die Typfunktionen

$$\begin{aligned} \texttt{add}_{\mathbb{T}} &:= \Lambda\delta :: *.\texttt{match}\ \delta\ \texttt{with}\ \ \alpha\times\beta\ \ \Rightarrow\ \ bop_{\mathbb{T}}[\texttt{OpPlus}\times\alpha\times\beta] \\ \texttt{average}_{\mathbb{T}} &:= \Lambda\delta :: *.\texttt{match}\ \delta\ \texttt{with}\ \ \alpha\times\beta\ \ \Rightarrow\ \ bop_{\mathbb{T}}[\texttt{OpDivide}\times\texttt{add}_{\mathbb{T}}[\alpha\times\beta]\times\texttt{int}] \end{aligned}$$

```
( add(x)(y)     = x + y,
  average(x)(y) = add(x)(y) / 2
)
```

$$\begin{aligned} \texttt{add}_{\mathbb{T}} &:= \Lambda\alpha :: *.\Lambda\beta :: *.bop_{\mathbb{T}}[\texttt{OpPlus} \times \alpha \times \beta] \\ \texttt{average}_{\mathbb{T}} &:= \Lambda\alpha :: *.\Lambda\beta :: *.bop_{\mathbb{T}}[\texttt{OpDivide} \times \texttt{add}_{\mathbb{T}}[\alpha][\beta] \times \texttt{int}] \end{aligned}$$

Abbildung 9.4: Einfaches EML-Skript (oben) und daraus abgeleitete Typfunktionen (unten).

zuordnen. Diese Typfunktionen lassen jedoch unberücksichtigt, daß EML-Superkombinatoren partiell appliziert werden können. Darüber hinaus ist das Überladen von Superkombinatoren in EML nicht erlaubt, so daß der Test auf einen kartesischen Typ entfallen kann. Abbildung 9.4 (unten) zeigt die entsprechend angepaßten Typfunktionen.

Es sollte klar sein, daß sich jedes EML-Skript in eine Menge von Gleichungen über Typfunktionen transformieren läßt. Wir verzichten auf die Angabe einer entsprechenden Übersetzungsfunktion, weil wir den abstrakten Interpreter auf der abstrakten Darstellung von EML (CoreEML) operieren lassen wollen. Die Abbildung auf Typfunktionen dient uns lediglich zur Motivation des Berechnungsverfahrens.

Liegt ein EML-Skript in transformierter Form vor, läßt sich der Ergebnistyp eines Programms recht einfach bestimmen. Zunächst wandelt man den auszuwertenden Term in einen Typterm um, indem man Konstanten durch ihr Typsymbol, Funktionssymbole durch Typfunktionssymbole und Wertapplikationen durch Typapplikationen ersetzt. So entsteht zum Beispiel aus dem Term `average(2)(3)` der Typterm $\texttt{average}_{\mathbb{T}}[\texttt{int}][\texttt{int}]$.

Diesen Term reduziert man durch gewöhnliche Termersetzung, wobei man eine *call by value* Semantik zu Grunde legt[1]:

$$\begin{aligned} \texttt{average}_{\mathbb{T}}[int][int] &\rightsquigarrow (\Lambda\alpha.\Lambda\beta.bop_{\mathbb{T}}[\texttt{OpDivide} \times \texttt{add}_{\mathbb{T}}[\alpha][\beta] \times \texttt{int}])[\texttt{int}][\texttt{int}] \\ &\rightsquigarrow (\Lambda\beta.bop_{\mathbb{T}}[\texttt{OpDivide} \times \texttt{add}_{\mathbb{T}}[\texttt{int}][\beta] \times \texttt{int}])[\texttt{int}] \\ &\rightsquigarrow bop_{\mathbb{T}}[\texttt{OpDivide} \times \texttt{add}_{\mathbb{T}}[\texttt{int}][\texttt{int}] \times \texttt{int}] \\ &\rightsquigarrow bop_{\mathbb{T}}[\texttt{OpDivide} \times (\Lambda\alpha.\Lambda\beta.bop_{\mathbb{T}}[\texttt{OpPlus} \times \alpha \times \beta])[\texttt{int}][\texttt{int}] \times \texttt{int}] \\ &\rightsquigarrow bop_{\mathbb{T}}[\texttt{OpDivide} \times (\Lambda\beta.bop_{\mathbb{T}}[\texttt{OpPlus} \times \texttt{int} \times \beta])[\texttt{int}] \times \texttt{int}] \\ &\rightsquigarrow bop_{\mathbb{T}}[\texttt{OpDivide} \times bop_{\mathbb{T}}[\texttt{OpPlus} \times \texttt{int} \times \texttt{int}] \times \texttt{int}] \\ &\rightsquigarrow bop_{\mathbb{T}}[\texttt{OpDivide} \times \texttt{int} \times \texttt{int}] \\ &\rightsquigarrow \texttt{int} \end{aligned}$$

Leider ist dieses Vorgehen für rekursive Superkombinatoren nicht praktikabel, da die Typfunktion zu `If` in allen drei Argumenten strikt ist (siehe Abbildung 9.2 auf Seite 179).

Betrachten wir zum Beispiel die Fakultätsfunktion `fac`, die daraus abgeleitete Typfunktion $\texttt{fac}_{\mathbb{T}}$ (siehe Abbildung 9.5) und den Term `fac(10)`, der dem Typausdruck $\texttt{fac}_{\mathbb{T}}[\texttt{int}]$ entspricht.

[1] Die Semantik entspricht im Wesentlichen der Reduktionsrelation $\rightsquigarrow$ für System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$-Terme.

```
( fac(x) = If(x==0, 1, x*fac(x-1) )
)
```

$$\texttt{fac}_{\mathbb{T}} \;:=\; \Lambda\alpha :: *.\; \texttt{If}_{\mathbb{T}}[\mathit{bop}_{\mathbb{T}}[\texttt{OpEq} \times \alpha \times \texttt{int}] \times \texttt{int} \times \mathit{bop}_{\mathbb{T}}[\texttt{OpMult} \times \alpha \times \texttt{fac}_{\mathbb{T}}[\,\mathit{bop}_{\mathbb{T}}[\texttt{OpSub} \times \alpha \times \texttt{int}]\,]\,]\,]$$

Abbildung 9.5: Fakultätsfunktion in EML und die daraus hergeleitete Typfunktion.

Der Ergebnistyp von `fac(10)` entspricht der Normalform zu $\texttt{fac}_{\mathbb{T}}[\texttt{int}]$. Um diese zu berechnen, ersetzen wir den Typparameter α in der rechten Seite von $\texttt{fac}_{\mathbb{T}}$ durch den Typ `int`. Da auch $\texttt{If}_{\mathbb{T}}$ strikt auszuwerten ist, wird zunächst das erste Argument reduziert. Wie man leicht nachvollzieht, entsteht dabei der Typ `bool`. Das zweite Argument kann nicht weiter reduziert werden, daher wird mit der Reduktion des dritten fortgefahren:

$$\begin{aligned} & \mathit{bop}_{\mathbb{T}}[\ \texttt{OpMult} \times \texttt{int} \times \texttt{fac}_{\mathbb{T}}[\ \mathit{bop}_{\mathbb{T}}[\texttt{OpSub} \times \texttt{int} \times \texttt{int}]\ ]\ ] \\ \rightsquigarrow\ & \mathit{bop}_{\mathbb{T}}[\ \texttt{OpMult} \times \texttt{int} \times \texttt{fac}_{\mathbb{T}}[\texttt{int}]\ ] \end{aligned}$$

Der nächste Redex ist der rekursive Aufruf von $\texttt{fac}_{\mathbb{T}}$ – wir drehen uns also im Kreis. Mit einfacher Termersetzung kommt man daher nicht weiter.

In Anlehnung an die Arbeit von Widera [178] wollen wir jetzt ein iteratives Verfahren zur Berechnung des Ergebnistyps entwickeln. Um Rekursion zu erkennen, merkt sich der iterative Prozeß, welche Funktionsaufrufe (Funktionsname und Argumenttypen) sich gerade in Bearbeitung befinden und zu welchen Ergebnistypen diese Aufrufe jeweils führen können.

Bevor wir das weitere Vorgehen beschreiben, erweitern wir die Relation $<:$ um ein kleinstes Element $\bot$ ($\forall T \in \mathcal{C}_{\mathbb{T}} : T <: \bot$) – der Wert $\bot$ bringt zum Ausdruck, daß wir keine Aussage über einen Typ machen können. Die Typfunktionen zu den Grundoperationen erweitern wir so, daß sie $\bot$ zum Ergebnis haben, wenn ein oder mehr Argumente vom Wert $\bot$ sind. Die Typfunktion zu `If` lassen wir unverändert.

Der iterative Prozeß (der abstrakte Interpreter) ist mit einem Zustand Ω ausgestattet. Ω ist eine Menge von Paaren aus Funktionsabschluß und einer Menge von möglichen Ergebnistypen, die sich beim Ausrechnen des Abschlusses ergeben können. Initial enthält diese Menge das Paar $(T, \Delta(T)_0)$, wobei T der zu typisierenden Funktionsaufruf ist, und $\Delta(T)_0$ als einziges Element $\bot$ enthält – wir können ja noch keine Aussage über den Ergebnistyp von T machen.

In Iterationsschritt i wird die Menge $\Delta(T)_i$ berechnet. Dazu wird $\Delta(T)_i$ mit $\Delta(T)_{i-1}$ initialisiert und es werden parallel alle Berechnungspfade durchlaufen. Die Typen, die die einzelnen Pfade zum Ergebnis haben können, werden zur Menge $\Delta(T)_i$ hinzugefügt. Bei der Berechnung eins `If`-Terms wird also nicht mit der Regel $\texttt{If}_{\mathbb{T}}$ reduziert. Die Ergebnistypen von `then`- und `else`-Zweig werden parallel berechnet und in $\Delta(T)_i$ mit aufgenommen. Bei der Reduktion eines Pfades können drei Situationen auftreten:

1. Der Pfad enthält keinen rekursiven Aufruf und ruft auch keine benutzerdefinierte Funktion auf. Der Ergebnistyp ergibt sich allein aus Grundoperationen bzw. deren Komposition und kann somit direkt berechnet und zur Menge $\Delta(T)_i$ hinzugenommen werden.

2. Der Pfad enthält einen rekursiven Aufruf; d.h. es gibt ein $(T', \Delta(T'))$ in Ω, so daß T' zum in Frage stehenden Funktionsaufruf paßt. Jedes Element aus $\Delta(T')$ ist ein möglicher Ergebnistyp von T'. Der zum Berechnungspfad gehörende Ausdruck muß unter Berücksichtigung aller dieser Typen ausgerechnet werden, um anschließend $\Delta(T)_i$ um die resultierenden Ergebnistypen zu erweitern.

3. Der Pfad enthält einen polymorph-rekursiven-, oder den Aufruf einer benutzerdefinierten Funktion. In diesem Fall wird ein neuer Prozeß gestartet, dessen initialer Zustand Ω' eine Erweiterung des Zustandes Ω um das Paar $(T_c, \{\bot\})$ ist, wobei T_c der in Frage stehende Aufruf ist.

Das Verfahren stoppt, wenn $\Omega_i = \Omega_{i-1}$ gilt. Die resultierende Menge der möglichen Ergebnistypen bezeichnen wir mit $\Delta(T)$.

Anmerkung 9.1. Das Verfahren entspricht im gewissen Sinne der Ausführung eines dynamisch getypten Programmes. Ist eine Typfunktion an einer Stelle nicht definiert, bricht das Verfahren mit einem Typfehler ab. $\Diamond$

Spielen wir das Verfahren am Beispiel der Fakultät einmal durch. Der abstrakte Interpreter startet im Zustand $\Omega = \{(\texttt{fac}[\texttt{int}], \{\bot\}$. Beide Pfade des `If`-Terms müssen berechnet werden. Der erste Pfad führt unmittelbar zum Typ `int`, der zweite enthält einen rekursiven Aufruf. Als einzig möglicher Ergebnistyp kommt $\bot$ in Frage. Da die abstrakte Multiplikation mit $\bot$ zum Ergebnis $\bot$ führt, resultiert der zweite Pfad in $\bot$. Nach der ersten Iteration befindet sich der Prozeß somit im Zustand $\Omega_1 = \{(\texttt{fac}[\texttt{int}], \{\bot, \texttt{int}\})\}$.

Im zweiten Iterationsschritt kann der rekursive Aufruf zum Wert `int` oder zu $\bot$ führen. Die abstrakte Multiplikation von `int` und `int` führt zum Wert `int` – die Menge der möglichen Ergebnistypen vergrößert sich also nicht mehr und es gilt $\Omega_1 = \Omega_2$, woraus $\texttt{fac}[\texttt{int}] = \texttt{int}$ folgt.

Das Fakultätsbeispiel war insofern einfach, als daß es, abgesehen von $\bot$, nur einen möglichen Ergebnistyp gibt. Was tun, wenn mehrere Ergebnistypen möglich sind?

In jedem Fall brauchen wir einen eindeutigen Ergebnistyp, da EML statisch getypt ist und durch Typelimination übersetzt werden soll. Das Supremum $\bigsqcup \Delta(T)$ bietet sich als gute Lösung an, da alle Typen in $\Delta(T)$ mit diesem Typ verträglich sind. Läßt sich kein Supremum finden, so liegt ein Typfehler vor und wir ordnen dem zugehörigen Term den Typ $\bot$ zu.

Wenn wir jetzt Programmcode für eine rekursive Funktion erzeugen, so gehen wir davon aus, daß jeder rekursive Aufruf einen Wert vom Typ $\bigsqcup \Delta(T)$ generiert. Da das Supremum aber nicht in $\Delta(T)$ liegen muß, kann es sein, daß wir bei der Typberechnung nicht alle Fälle berücksichtigt haben.

Betrachten wir exemplarisch eine rekursive Funktion f, die mit einem Argument x vom Typ T aufgerufen wird und sei $\bigsqcup \Delta(f_{\mathbb{T}}[T]) \notin \Delta(f_{\mathbb{T}}[T])$. Steht ein rekursiver Aufruf in einem bestimmten Kontext (z.B. $y : T' * f(x : T)$), so haben wir bei der Berechnung des Ergebnistyps zu $f(x)$ nicht geprüft, ob die Typfunktion $bop_{\mathbb{T}}$ an der Stelle $\texttt{OpMult} \times T' \times \bigsqcup \Delta(f_{\mathbb{T}}F[T])$ definiert ist. Außerdem könnte $bop_{\mathbb{T}}[\texttt{OpMult} \times T' \times \bigsqcup \Delta(f_{\mathbb{T}}[T])]$ zu einem Typ führen, der nicht zum Supremum paßt (weil er größer, oder mit dem Supremum unvergleichbar ist).

Eine naive Lösung wäre, zu verlangen, daß das Supremum grundsätzlich in $\Delta(T)$ liegen muß. Dieser Ansatz ist allerdings sehr restriktiv, wie wir am Beispiel der Typfunktion zu `IF` gezeigt haben.

Eine bessere Strategie ist, in einem zweiten Gang nochmal jeden Berechnungspfad zu durchlaufen, wobei man jeweils $\bigsqcup \Delta(T)$ als einzig mögliches Ergebnis der Rekursion betrachtet. Erhält man dabei einen Ergebnistyp T' mit $T' <: \bigsqcup \Delta(T)$, so ist $\bigsqcup \Delta(T)$ der gesuchte Ergebnistyp, ansonsten $\bot$.

Das soeben beschriebene Verfahren der Ergebnistypberechnung ist aufwendig und seine Komplexität (Laufzeit) hängt stark von der Größe der Menge $\Delta(T)$ ab.

Da uns die Aspekte des Typsystems nur am Rande interessieren, nehmen wir bei der Implementierung des abstrakten Interpreters folgende Vereinfachung vor: Anstatt alle möglichen Ergebnistypen in Ω zu speichern, berechnen wir das Supremum iterativ. Läßt sich zu zwei Typen kein Supremum finden, brechen wir die Typberechnung mit dem Ergebnis $\bot$ ab.

Es ist klar, daß wir damit einige typisierbare Funktionen ausgrenzen, denn offenbar gilt

$$\exists \bigsqcup \{\alpha_1, \alpha_2, \alpha_3\} \Rightarrow \exists \bigsqcup \{\alpha_1, \alpha_2\}$$

im Allgemeinen nicht (siehe Abbildung 9.6). Diese Einschränkung ist aber insofern vertretbar, als daß sie nur bestimmte Formen der Mehrfachvererbung betrifft.

Ebenfalls verzichten wir auf die explizite Programmierung des Tests, ob der Kontext eines rekursiven Aufrufs mit dem ermittelten Supremum verträglich ist. Später, nach der Übersetzung von CoreEML nach C++ , wird ein eventueller Fehler ohnehin vom C++ -*Typechecker* erkannt.

In letzter Konsequenz bedeuten diese Vereinfachungen, daß wir `If`-Terme nicht mehr gesondert behandeln müssen, sondern sofort mit $\texttt{If}_{\mathbb{T}}$ reduzieren können. Typmengen sind nicht mehr erforderlich und im Zustand des Interpreters muß nur noch die Aufrufspur gespeichert sein.

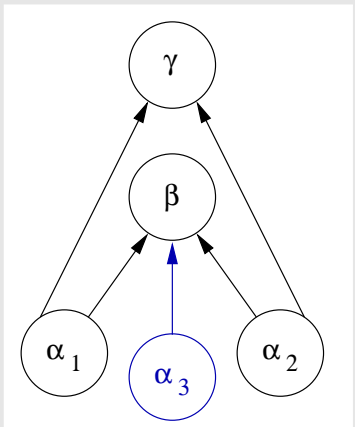


Abbildung 9.6: Grafische Darstellung der Typrelation <:. Gilt $\alpha <: \beta$, so sind α und β mit einer gerichteten Kante verbunden, die von α ausgeht. Der reflexive Anteil von <: ist nicht dargestellt. Offenbar existiert kein Supremum zu α_1 und α_3. Zwar sind sowohl β, als auch γ obere Schranken zu $\{\alpha_1, \alpha_3\}$, allerdings sind diese bzgl. <: unvergleichbar, so daß sich kein kleinstes Element bestimmen läßt. Zur Menge $\{\alpha_1, \alpha_2, \alpha_3\}$ läßt sich im Verlgeich dazu, mit β sehr wohl ein Supremum finden.

9.3 Typinferenz für EML-Skripte

Im vorherigen Abschnitt haben wir ein Verfahren zur Berechnung des Ergebnistyps von EML-Programmen beschrieben. Zentraler Bestandteil war ein abstrakter Interpreter für EML. In diesem Abschnitt wollen wir eine Implementierung des Typinferenzverfahrens beschreiben, die direkt auf der internen Darstellung von EML-Programmen aufsetzt; mit anderen Worten werden wir einen abstrakten Interpreter für CoreEML vorstellen, der auf Typsymbolen operiert.

Den Interpreter werden wir nicht als System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$-Funktion, sondern in Form von Inferenzregeln angeben, da uns diese eine kompaktere und übersichtlichere Schreibweise erlauben. Wir werden später kurz darauf eingehen, wie man aus dem Regelwerk eine System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$-Funktion gewinnt kann und wie man diese in C++ implementiert.

Wir setzen voraus, daß ein EML-Skript $\mathcal{P}_{EML}$ als CoreEML-Skript $\mathcal{P}_{CoreEML}$ vorliegt.

Der Zustand des abstrakten Interpreters besteht aus zwei Komponenten:

- einem Stack, auf dem die Argumente an einen Superkombinator abgelegt werden; sowie
- einer Menge aus den in Bearbeitung befindlichen Superkombinatoraufrufen (Funktionsabschlüsse).

Den Stack realisieren wir als Polytypliste und notieren ihn in der Form $\{\!|T_1 : \cdots : T_n|\!\}$, wobei die Stackspitze immer links steht. Die Aufrufhistorie stellen wir als Menge von Stacks dar.

Die Zustandsübergänge des Interpreters beschreiben wir im Stil einer *big step* Semantik.

$$\frac{\text{Prämisse}}{\{\!|\alpha_s|\!\} \quad ct \Downarrow_{\mathbb{T}} \{\!|\alpha'_s|\!\} \quad ct'} \quad \textbf{(Regelname)}$$

Die Regel **Regelname** beschreibt den Übergang vom Zustand $\{\!|\alpha_s|\!\} \quad ct$ in den Zustand $\{\!|\alpha'_s|\!\} \quad ct'$. Der Übergang ist möglich, wenn der aktuelle Stack mit dem Muster α_s unifizierbar ist, und die in der Prämisse genannten Bedingungen erfüllbar sind.

Der Interpreter erreicht einen Endzustand, wenn der Stack ausschließlich eine Konstante oder eine Variable enthält.

Abbildung 9.7 listet alle Transitionsreglen auf.

Das Regelwerk ist algorithmisch zu interpretieren. In einem bestimmten Zustand sucht man eine Regel, deren Stackmuster zum aktuellen Stack paßt. Anschließend wird die Regel instanziiert; d.h. Mustervariablen werden durch konkrete Typen ersetzt. Wie man sich leicht überzeugen kann, lassen sich die Muster so ordnen, daß dieser Vorgang durch einen `Match`-Term realisiert werden kann.

Hat man eine passende Regel gefunden, so werden die Prämissen von oben nach unten und von links nach rechts abgearbeitet. Sofern eine Prämisse einen Zustandsübergang beschreibt, wird die linke Seite als neuer Zustand angenommen, um daraus einen Endzustand zu berechnen. Die Prämisse gilt als erfüllt, wenn der berechnete Endzustand zum Muster der rechten Seite paßt.

Eine Prämisse kann aber auch allgemeine Forderungen enthalten, die in der Implementierung z.B. durch den Einsatz von `If` oder `Match` abgefragt werden können. Zum Beispiel wird in der

$\boxed{\Downarrow_{\mathbb{T}}}$ CoreEML

$$\frac{kopf(\alpha_T) \notin \{\texttt{App}, \texttt{Var}, \texttt{PaSc}\}}{\{\!|\alpha_T : \alpha_S|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|\alpha_T : \alpha_S|\!\}\ \ ct} \quad \textbf{(Const)}$$

$$\{\!|\texttt{Var}(p) : \alpha_s|\!\}\ \ ct \Downarrow_{\mathbb{T}} \{\!|\texttt{Var}(p) : \alpha_S|\!\}\ \ ct \quad \textbf{(FVar)}$$

$$\frac{\begin{array}{c}\{\!|\alpha_A|\!\}\ \ ct \Downarrow_{\mathbb{T}} \{\!|T_A|\!\}\ \ ct \\ \{\!|\alpha_F : T_A : \alpha_S|\!\}\ \ ct \Downarrow_{\mathbb{T}} \{\!|T : \alpha'_S|\!\}\ \ ct\end{array}}{\{\!|\texttt{App}(\alpha_F, \alpha_A) : \alpha_S|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|T : \alpha'_S|\!\}\ \ ct} \quad \textbf{(App)}$$

$$\frac{\texttt{bop}_{\mathbb{T}}[\eta \times \alpha_1 \times \alpha_n] = T}{\{\!|\eta : \alpha_1 : \alpha_2 : \alpha_S|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|T : \alpha_S|\!\}\ \ ct} \quad \textbf{(BinaryOp}_\eta\textbf{)}$$

falls $\eta \in \{\texttt{OpPlus}, \texttt{OpMinus}, \texttt{OpMult}, \texttt{OpDivide}, \texttt{OpEqual}, \cdots\}$

$$\frac{\texttt{uop}_{\mathbb{T}}[\eta \times \alpha] = T}{\{\!|\eta : \alpha : \alpha_S|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|T : \alpha_S|\!\}\ \ ct} \quad \textbf{(UnaryOp}_\eta\textbf{)}$$

mit $\eta \in \{\texttt{OpUnaryPlus}, \texttt{OpUnaryMinus}, \texttt{OpNot}\}$

$$\frac{\texttt{If}_{\mathbb{T}}[\alpha_C \times \alpha_T \times \alpha_E] = T}{\{\!|\texttt{OpIf} : \alpha_C : \alpha_T : \alpha_E : \alpha_S|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|T : \alpha_S|\!\}\ \ ct} \quad \textbf{(If)}$$

$$\frac{\begin{array}{c}v(v_1)\cdots(v_n) = t \in \mathcal{P}_{EML}\ \ \vdash t : T, v : \texttt{Var}(\alpha_V) \\ \forall i \in \{1, \cdots, n\} : \vdash v_i : \texttt{Var}(T_i)\ \ \vee\ \ \vdash v_i : \texttt{App}(\texttt{OpNot}, \texttt{Var}(T_i)) \\ \{\!|\texttt{Var}(\alpha_V) : \alpha_n : \cdots : \alpha_1|\!\} \notin ct \\ \{\!|T[\alpha_1/\texttt{Var}(T_1), \cdots, \alpha_n/\texttt{Var}(T_n)]|\!\}\ \ ct \cup \{\{\!|\texttt{Var}(\alpha_V) : \alpha_1 : \cdots : \alpha_n|\!\}\} \Downarrow_{\mathbb{T}} \{\!|T|\!\}\ \ ct'\end{array}}{\{\!|\texttt{Var}(\alpha_V) : \alpha_1 : \cdots : \alpha_n : \alpha_S|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|T : \alpha_S|\!\}\ \ ct} \quad \textbf{(Sc)}$$

$$\frac{\begin{array}{c}v(v_1)\cdots(v_n) = t \in \mathcal{P}_{EML}\ \ \vdash t : T, v : \texttt{Var}(\alpha_V) \\ \{\!|\texttt{Var}(\alpha_V) : \alpha_1 : \cdots : \alpha_n|\!\} \in ct\end{array}}{\{\!|\texttt{Var}(\alpha_V) : \alpha_1 : \cdots : \alpha_n : \alpha_S|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|\bot : \alpha_S|\!\}\ \ ct} \quad \textbf{(ScRek)}$$

$$\frac{\begin{array}{c}v(v_1)\cdots(v_n) = t \in \mathcal{P}_{EML}\ \ \vdash t : T, v : \texttt{Var}(\alpha_V) \\ n > m\end{array}}{\{\!|\texttt{Var}(\alpha_V) : \alpha_1 : \cdots : \alpha_m|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|\texttt{PaSc}(\texttt{Var}(\alpha_V), \{\!|\alpha_1 : \cdots : \alpha_m|\!\})|\!\}\ \ ct} \quad \textbf{(ScPa)}$$

$$\frac{\{\!|\texttt{Var}(\alpha_V) : \alpha_1 : \cdots : \alpha_m : \beta)|\!\}\ \ ct \Downarrow_{\mathbb{T}} \{\!|T' : s'|\!\}\ \ ct}{\{\!|\texttt{PaSc}(\texttt{Var}(\alpha_V), \{\!|\alpha_1 : \cdots : \alpha_m|\!\}) : \beta|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|T' : s'|\!\}\ \ ct} \quad \textbf{(ScPaApp)}$$

$$\frac{\{\!|T_{inst}|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|T'|\!\}\ \ ct}{\{\!|\texttt{OpLet} : \texttt{Let}(\{\!|\texttt{Var}(\alpha_1) = T_1, \cdots, \texttt{Var}(\alpha_n) = T_n|\!\}, T) : \alpha_S|\!\}\ \ ct\ \ \Downarrow_{\mathbb{T}} \{\!|T' : \alpha_S|\!\}\ \ ct} \quad \textbf{(Let)}$$

mit $T_{inst} = T\,[T_1/\texttt{Var}(\alpha_1), T'_2/\texttt{Var}(\alpha_2), \cdots, T'_n/\texttt{Var}(\alpha_n)]$
$T'_2 = T_2\,[T_1/\texttt{Var}(\alpha_1)]$
$T'_3 = T_3\,[T_1/\texttt{Var}(\alpha_1), T'_2/\texttt{Var}(]$
$\cdots$
$T'_n = T_n\,\left[T_1/\texttt{Var}(\alpha_1), T'_2/\texttt{Var}(\alpha_2), \cdots T'_{n-1}/\texttt{Var}(T_{n-1})\right]$

Abbildung 9.7: Typberechnung

Prämisse der Regel **Const** verlangt, daß der an die Mustervariable α_T gebundene Typ weder ein Applikations-, noch ein partieller Applikationsknoten, noch eine funktionale Variable ist.

Betrachten wir die Regeln im einzelnen:

Liegt auf der Stackspitze eine Konstante, also ein C++ -Typname, dessen Kopfsymbol weder `App`, noch `Var` oder `PaSc` ist, sind keine weiteren Berechnungen erforderlich. Dasselbe gilt für funktionale Variablen, sofern diese keine Operationssymbole oder Superkombinatoren bezeichnen (und somit unter eine der Regeln **BinaryOp**, **UnaryOp**, **If**, **Sc** oder **SCRek**, oder **ScPa** fallen).

Befindet sich auf der Stackspitze ein Knotentyp $\texttt{App}(\alpha_F, \alpha_A)$, wird zunächst das Argument reduziert (strikte Auswertung). Dazu wird der Interpreter mit einem Stack gestartet, der alleine das Argument α_A enthält. Anschließend wird der Funktionsaufruf reduziert. Auch dazu wird der Interpreter mit einem neuen Stack gestartet. Dieser geht aus dem alten Stack hervor, indem man den Applikationsknoten entfernt und dafür nacheinander das ausgewertete Argument T_A und die Funktion α_F auf dem Stack ablegt. Kann dieser Stack zu einem Stack $\{\!|T : \alpha'_S|\!\}$ reduziert werden, kann der Automat den Applikationsknoten durch den Typ T ersetzen und mit der Berechnung fortfahren.

Abbildung 9.8 verdeutlicht die Wirkung der Regel **App**. Es werden zunächst alle Argumente ausgewertet und auf dem Stack abgelegt, bis schließlich ein Funktionssymbol auf der Stackspitze liegt.

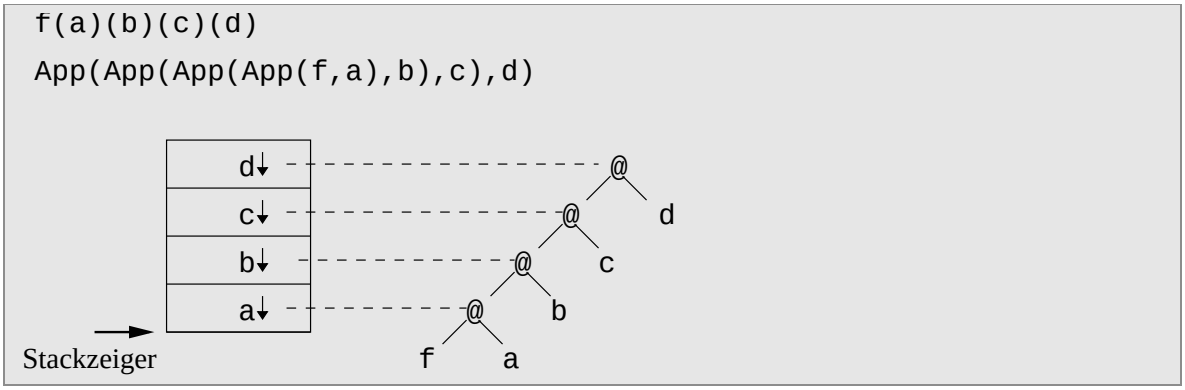


Abbildung 9.8: Wirkung der Regel **App**.

Eingebaute Funktionen werden von den Regeln $\texttt{BinaryOp}_\eta$, $\texttt{UnaryOp}_\eta$ und `If` abgedeckt. Sofern genügend Funktionsargumente auf dem Stack liegen, werden diese als Parameter an die zugehörigen Typfunktionen übergeben und vom Stack gelöscht.

Genau genommen bezeichnen $\texttt{BinaryOp}_\eta$ und $\texttt{UnaryOp}_\eta$ Mengen von Regeln, wobei η ein Platzhalter für eine CoreEML-Variable ist, die einem eingebauten EML-Operator zugeordnet ist (siehe Abbildung 8.4 auf Seite 164). Zum Beispiel schreiben wir $\texttt{BinaryOp}_{\texttt{OpPlus}}$, wenn wir die Reduktion mit dem binären Plus-Operator meinen.

Die Regeln **Sc**, **ScRek** und **ScPa** zur Behandlung von Superkombinatoraufrufen sind etwas aufwendiger. Liegt auf der Stackspitze eine funktionale Variable $\texttt{Var}(\alpha_V)$, die im EML-Skript $\mathcal{P}_{EML}$ als Superkombinator verwendet wird[2], sind drei Fälle zu unterscheiden:

[2]Wir geben die Superkombinatordefinition der Lesbarkeit halber als EML-Term an. Bedingungen der Art $\vdash t : T$ beziehen sich weniger auf die Typinferenz, sondern geben uns Zugriff auf die Darstellung von t als

- Es liegen genügend viele Argumente für den Superkombinator auf dem Stack und es handelt sich um keinen rekursiven Aufruf (Regel **Sc**). In diesem Fall werden Funktionsvariable und Argumente vom Stack entfernt, die instanziierte rechte Seite der Superkombinatordefinition wird auf dem Stack abgelegt, die Aufrufspur entsprechend erweitert, und anschließend eine Normalform T berechnet. War das erfolgreich möglich, können Funktionsname und Argumente vom Stack entfernt und durch den Typ T ersetzt werden.

- Es liegen zwar genügend viele Argumente auf dem Stack, jedoch handelt es sich um einen rekursiven Aufruf (Regel **ScRek**). Dann werden Funktionssymbol und Argument entfernt und der Typ $\perp$ auf dem Stack abgelegt.

- Es handelt sich um eine partielle Applikation (Regel **ScPa**). In dieser Situation werden Funktionsname und Argumente durch einen `PaSc`-Knoten ersetzt, der den Funktionsnamen und die bisher applizierten Argumente speichert.

Knoten, die aus partieller Applikation entstanden sind, können auf weitere Argumente angewendet werden (Regel **ScPaApp**). Liegen weitere Argumente vor, wird der `PaSc`-Knoten aufgebrochen; d.h. er wird vom Stack entfernt, um anschließend die in ihm gespeicherten Argumente und den Funktionsnamen auf dem Stack abzulegen. Anschließend wird mit der Reduktion fortgefahren, wobei der Typ T' entsteht.

`Let`-Ausdrücke werden von der Regel **Let** abgedeckt. Die Regel ist so konstruiert, daß einmal eingeführte `Let`-Variablen in nachfolgenden `Let`-Bindings verwendet werden können. Ansonsten wird als Ergebnis der Term T' zurückgeliefert, der sich durch Reduktion des instanziierten Terms T_{inst} ergibt.

In Anhang A.4 wird die Arbeitsweise des abstrakten Interpreters am Beispiel der Fakultätsfunktion demonstriert.

9.4 Implementierung der Typberechnung

Die Untertyprelation haben wir bei der Vorstellung des abstrakten Interpreters implizit vorausgesetzt. Da es in C++ keine direkte Möglichkeit gibt, zu prüfen, ob zwei Typen in dieser Relation stehen, müssen wir ein entsprechendes Metaprogramm selbst zur Verfügung stellen.

Am einfachsten scheint die Implementierung von <: als Prädikat:

```
template <typename T1,typename T2> struct
subType { typedef FALSE RET }; };
template <> struct
subType<signed char,signed int> { typedef TRUE RET; };
// ...
```

Das reflexive und transitive Verhalten von <: muß dabei explizit berücksichtigt werden, so daß man sehr rasch zu einer hohen Zahl an Template-Spezialisierungen kommt und die Erweiterung von <: um neue Typen sehr aufwendig wird.

CoreEML-Term. Es ist darauf zu achten, daß ein Superkombinator mit einer Striktheitsannotation versehen sein kann.

```
template <typename T,typename E>
struct supremum;

template <typename T> struct supremum<T,T>       { typedef T RET; };
template <typename T> struct supremum<TBottom,T> { typedef T Ret; };
template <typename T> struct supremum<T,TBottom> { typedef T Ret; };
template <> struct supremum<int,double>          { typedef double RET; };

template <typename T,typename E> struct
inferIf<bool,T,E> { typedef typename supremum<T,E>::RET RET; };
```

Abbildung 9.9: Metaprogramm zur Berechnung des Supremums zweier Typen.

Eine Metafunktion, die anhand von `subType` das Supremum zweier Typen T_1 und T_2 berechnet, läßt sich nur sehr schwer und unter großem Aufwand implementieren. Zunächst müßte man alle Typen heraussuchen, die größer als T_1 und T_2 sind, um dann das eindeutig kleinste Element dieser Menge zu bestimmen. Das Problem liegt dabei in der Konstruktion der Menge aller Typen, die in einer Übersetzungseinheit verwendet werden, da diese von C++ -Programm zu C++ -Programm variiert.

Anstatt <: direkt zu implementieren legen wir deshalb eine Metafunktion `supremum` fest, die das Supremum zweier Typen direkt berechnet (siehe Abbildung 9.9). Den kleinsten Typ $\bot$ stellen wir als Klasse `Bottom` dar, deren Konstruktor wir als privat vereinbaren, so daß man keine Objekte von diesem Typ erzeugen kann.

Abgesehen von Grundtypen, kann man die Berechnung des Supremums in vielen Fällen auch auf Konvertierbarkeit zurückführen. Sind Objekte einer Klasse `A` in Objekte einer Klasse `B` konvertierbar (aber nicht umgekehrt), so liegt die Vermutung nahe, daß `A` <: `B` gilt. Die Existenz von Konvertierungsoperatoren läßt es diesem Verfahren natürlich an Beweiskraft fehlen und es sollte daher nur optional angeboten werden.

In Anhang B.4 zeigen wir ein Metaprogramm zum Test auf Konvertierbarkeit.

Unter Verwendung der Metafunktion `supremum` lassen sich die Metafunktionen zur Berechnung der Rückgabetypen der in EML eingebauten Operationen sehr leicht formulieren (siehe Abbildung 9.10).

Die Transitionen des abstrakten Interpreters implementieren wir als Metafunktion `inferType`, die Stack, Aufrufspur und auszuwertenden Term als Parameter übernimmt und in `RET` den berechneten Ergebnistyp zurückliefert.

Im Vergleich zum Modell liegt der auszuwertende Term nicht auf der Stackspitze, sondern wird in einem separaten Argument übergeben – das macht das *pattern matching* ein wenig übersichtlicher. Stack und Aufrufspur werden wie im Modell als Polytyplisten dargestellt, die – wie gewohnt – mit Hilfe des Klassentemplates `TML::C` und der Klasse `TML::N` gebildet werden. Funktionsabschlüsse werden als Tupel mit Namen `aCall` dargestellt, wobei die erste Komponente den Namen des Superkombinators enthält und die zweite der Aufnahme der Argumentliste dient:

```
template <typename SCNAME, typename ARGS>
struct aCall {};
```

```
template <typename A1,typename A2,typename BinOp>
struct binaryReturn;

// OpPlus
template <typename T> struct
binaryReturn<T,T,OpPlus>       { typedef T RET;       };
template <typename T> struct
binaryReturn<TBottom,T,OpPlus> { typedef TBottom Ret; };
template <typename T> struct
binaryReturn<T,TBottom,OpPlus> { typedef TBottom Ret; };

template <typename T1,typename T2> struct
binaryReturn<T1,T2,OpPlus> { typedef typename supremum<T1,T2>::RET RET; };
// usw.

template <typename A,typename Op>
struct unaryReturn;

template <typename A> struct
unaryReturn<A,OpUnaryPlus> { typedef A RET; };
// usw.

template <typename C,typename T,typename E>
struct ifReturn;
template <typename T,typename E>
struct ifReturn<bool,T,E> { typedef supremum<T,E>::RET RET; };
```

Abbildung 9.10: Ausschnitt aus der Metafunktion `binaryReturn` zur Berechnung des Ergebnistyps von binären Operationen.

Jede Regel aus Abbildung 9.7 entspricht genau einer Template-Spezialisierung. Exemplarisch zeigen wir hier nur die Umsetzung der Regeln **App** und **Sc**.

Die Regel `App` kann angewendet werden, wenn ein `App`-Knoten auf der Stackspitze liegt. Die erste `typedef`-Deklaration entspricht der ersten Prämisse der Regel **App**. Sie wirkt wie ein `let`-Ausdruck für Typen: Der „Variablen" `Arg` wird das Ergebnis der abstrakten Interpretation des Arguments zugewiesen. Die zweite `typedef`-Deklaration entspricht der zweiten Prämisse der Regel **App**. Das ausgewertete Argument wird auf dem Stack abgelegt, um anschließend den abstrakten Wert der Funktionsapplikation zu berechnen und ihn im Feld `RET` verfügbar zu machen.

Am aufwendigsten ist die Behandlung von Superkombinatorapplikationen (siehe Abbildung 9.11), bei denen wir auf zahlreiche Standardlistenoperationen zurückgreifen: `Length<L>` berechnet die Länge der Liste `L`; `Take<N,L>` extrahiert die ersten `N` Elemente einer Liste `L`; `Drop<N,L>` löscht die ersten `N` Elemente aus der Liste `L` und `Member<E,L>` prüft das Vorhandensein des Elementes `E` in der Liste `L`. Die Implementierung dieser Funktionen findet sich in Anhang B.2.

Die ersten drei Zeilen entsprechen wieder einem `let`-Ausdruck: Der Variablen `Arity` wird die

```
template <typename SCRIPT,typename STACK,typename CALLTRACE,
          typename F,typename A> struct
inferType<SCRIPT,STACK,CALLTRACE,App<F,A> > {
  // Typ des Arguments errechnen und in Arg zwischenspeichern
  typedef typename
  inferType< SCRIPT, TML::N, CALLTRACE, A>::RET Arg;
  // Arg auf dem Stack ablegen und anschließend
  // Typ der Funktion errechnen ..
  typedef typename
  inferType< SCRIPT, TML::C<Arg, STACK>, CALLTRACE, F>::RET RET;
};

template <typename SCRIPT,typename STACK,typename CALLTRACE,int NAME> struct
inferType<SCRIPT, STACK, CALLTRACE, Var<NAME> > {
  // Aufzählungstypen können in integer-Werte konvertiert werden!
  enum { Arity       = getScArity< SCRIPT, Var<NAME> >::Ret };
  typedef typename TML::Take<Arity,STACK>::RET  ArgList;
  enum { isRecursiveCall = TML::Member<aCall<Var<NAME>, ArgList>,
                                       CALLTRACE
                                      >::RET };
  typedef If< isRecursiveCall,
              TBottom,
              typename
              If< Arity <= TML::Length<STACK>::Ret,
                  typename
                  inferType< SCRIPT,
                             typename TML::Drop<Arity,STACK>::RET,
                             TML::C< thisCall, CALLTRACE>,
                             typename
                             instantiate<SCRIPT,STACK,NAME,ArgList>::RET
                            >::RET,
                  PaSc<NAME, ArgList>
            >::RET RET;
};
```

Abbildung 9.11: Abstrakte Interpretation von Applikationsknoten (Regel **App**) und Superkombinatorapplikationen (Regeln `Sc`, `ScRek` und `PaSc`).

Zahl der Argumente zugewiesen, die der Superkombinator erwartet; `ArgList` nimmt die ersten `Arity` Elemente des Stacks auf und `isRecursiveCall` speichert das Ergebnis der Suche von `aCall<Var<NAME>, ArgList>` in der Aufrufspur.

Abhängig davon, ob es sich um einen rekursiven Aufruf handelt und genügend Argumente auf dem Stack bereitliegen, wird entweder $\perp$ zurückgegeben, der abstrakte Wert des instanziieren Superkombinators berechnet, oder aber ein `PaSc`-Knoten zurückgegeben.

Die Funktion `instantiate` ersetzt die in der rechten Seite einer Superkombinatordefinition auftretenden Variablen durch die entsprechenden Werte auf dem Stack.

Abbildung 9.12 zeigt einen Ausschnitt der Metafunktion `instantiate`. Der zu einem Term gehörende Baum wird traversiert: Es wird jeweils zu allen Kindknoten verzweigt, um dann *bottom-up* den neuen Term zu konstruieren. Wird beispielsweise ein Applikationsknoten besucht, so werden zunächst Funktions- und Argumentteil instanziiert, um dann einen neuen Applikationsknoten zu konstruieren und als Ergebnis an den Aufrufer zurückzuliefern. Trifft man beim Traversieren auf eine Variable, so muß diese durch den auf dem Stack befindlichen Wert ersetzt werden. Dabei ist darauf zu achten, daß der zu oberst liegende Wert dem letzten Argument an einen Superkombinator entspricht.

```
template <typename SCRIPT,typename STACK,int NAME,int EXPRESSION> struct
// "Konstanten"
instantiate { typedef EXPRESSION RET; }
template <typename SCRIPT,typename STACK,int NAME,
          typename F,typename A> struct
// Applikationsknoten
instantiate<SCRIPT,NAME,App<F,A> > {
  typedef typename instantiate<SCRIPT,NAME,F>::RET instF;
  typedef typename instantiate<SCRIPT,NAME,A>::RET instA;
  typedef App<insF,instA> RET;
};

template <typename SCRIPT,typename STACK,int NAME,
          int VARNAME> struct
instantiate<SCRIPT,STACK,NAME,Var<VARNAME> > {
  // Superkombinatordefinition extrahieren
  typedef typename GetScDefinition<SCRIPT,NAME>::RET SCDef;
  // Linke Seite dieser Definition in Lhs speichern.
  typedef typename SCef::Lhs_t Lhs;
  enum { // Zahl der Argumente, die Superkombinator erwartet
         arity    = GetScArity<SCRIPT,NAME>::Ret,
         // Position dieses Arguments in der Definition
         argpos   = GetArgPos<LHS,Var<VARNAME> >::Ret,
         // Position des Arguments auf dem Stack
         stackpos = arity - argpos - 1,
         stacksize = TML::Length<STACK>::RET
       };
  typedef typename
  IF< (argpos >= arity),// Falls argpos >= arity, handelt es sich nicht
                        // um ein Superkombinatorargument und wir lassen
      Var<VARNAME>,     // die Variable unverändert.
      typename
      IF< (stacksize <= stackpos), // Ist Argument wirklich da?
          TML::At<STACK,stackpos>::RET,
          Var<VARNAME>
        >::RET
    >::RET  RET;
};
```

Abbildung 9.12: Instanziierung eines Superkombinators für die Fälle, wo der zu instanziierende Term keine Variable ist.

Kapitel 10

Graphreduktion - Dynamische Semantik von EML

Der Semantik von EML liegt eine nicht-strikte Argumentauswertung zu Grunde. Funktionsargumente werden also nicht, wie beim gerade besprochenen abstrakten Interpreter, *vor* dem Funktionsaufruf ausgewertet, sondern unausgewertet als Funktionsabschluß übergeben. Um doppelte Argumentauswertung zu vermeiden, realisieren wir eine verzögerte Auswertungsstrategie mittels Graphreduktion [175]. Dabei lehnen wir uns an die von Peyton Jones und Lester in [140] vorgestellte *Template Instantiation Machine* (TIM) an[1].

Nachdem wir das Prinzip der Graphreduktion erläutert haben, werden wir die TIM vorstellen, um dann einen Überblick über ihre Implementierung zu geben.

10.1 Prinzip der Graphreduktion

Terme werden bei der Graphreduktion nicht als Bäume, sondern als Graphen dargestellt; d.h. mehrere Knoten können sich einen Kindknoten teilen. Bei der Reduktion wird immer der am weitesten links außen stehende Redex gewählt. Die Termreduktion entspricht dem Ersetzen eines Teilgraphen durch das Reduktionsergebnis. Handelte es sich um einen gemeinsamen Knoten, wird das Ergebnis für alle Vaterknoten sichtbar – dieser Effekt ist die Essenz der verzögerten Auswertung.

Betrachten wir die Arbeit eines Graphreduzierers an einem einfachen Beispiel: Gegeben sei das folgende EML-Programm

```
( square(x) = x * x,
  main()    = square(square(3))
)[ main ]
```

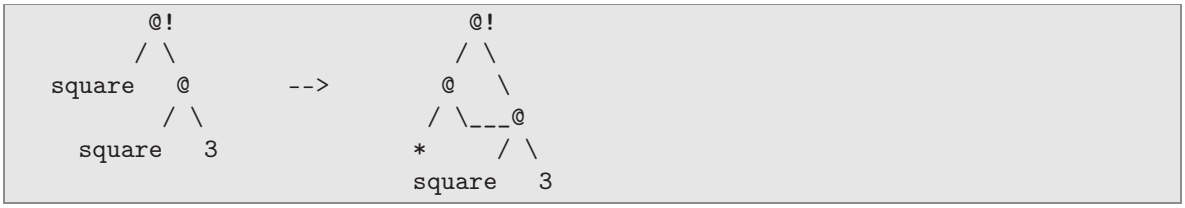
Der auszuwertende Term ist `main`, der durch folgenden, trivialen Graph repräsentiert wird:

[1] Anmerkung: Der Begriff *Template* hat nichts mit C++ -Templates zu tun. Gemeint ist hier die rechte Seite einer Superkombinatordefinition, die mit den konkreten Argumenten instanziiert wird.

Da der Superkombinator `main` keine Argumente erwartet, ist `main` selbst ein Redex und kann unmittelbar durch die rechte Seite der Superkombinatordefinition ersetzt werden. Dabei entsteht der Graph

Applikationsknoten stellen wir mit dem Symbol `@` dar.

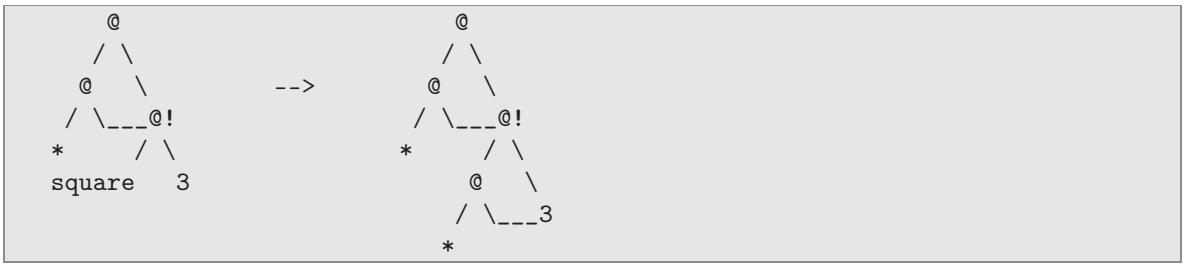
Der am weitesten links außen liegende Redex ist die äußere Anwendung des Superkombinators `square`. Man reduziert diese Superkombinatorapplikation, indem man den Wurzelknoten des Redex durch eine Instanz der rechten Seite von `square` ersetzt. Bei der Konstruktion des zugehörigen Graphen wird der Parameter `x` mit einem Zeiger auf den Teilgraphen substituiert, der das an `x` zu bindende Argument `square 3` repräsentiert:



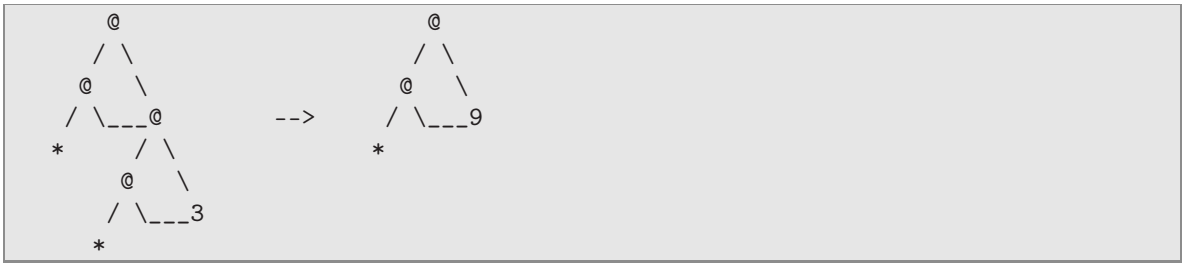
Den Wurzelknoten, der durch die Reduktion überschrieben wird, markieren wir mit einem Ausrufungszeichen.

Am resultierenden Graphen kann man den Unterschied zwischen einer einfachen, nicht-strikten Auswertungsstrategie und der *lazy evaluation* (Graphreduktion) klar erkennen: Dadurch daß im Rumpf des Superkombinators Zeiger und keine Bäume substituiert werden, taucht das Argument `square(3)` nicht doppelt auf, sondern wird zu einem gemeinsam genutzten Teilbaum – aus dem Baum ist ein echter Graph entstanden.

Der nächste Redex ist `square(3)`. Obwohl weiter außen stehend, kommt die Multiplikation als Redex nicht in Frage, da sie beide Funktionsargumente in ausgewerteter Form erwartet (`*` ist strikt in beiden Argumenten). Zur Reduktion von `square 3` wird abermals die rechte Seite der Superkombinatordefinition von `square` instanziiert, um anschließend den Wurzelknoten des Redex mit dem Ergebnis der Instanziierung zu überschreiben.



Die Konstante `3` wird zu einem gemeinsamen Knoten. Da die Argumente der inneren Multiplikation nicht weiter reduziert werden können, wird der Teilgraph zu `3 * 3` mit dem Multiplikationsergebnis ersetzt:



Das Ergebnis der Multiplikation wird in beiden Applikationsknoten der äußeren Multiplikation sichtbar. Im Gegensatz zur einer nicht-strikten Auswertungsstrategie ohne verzögerte Auswertung wird die innere Multiplikation also nur einmal durchgeführt. Der finale Reduktionsschritt ist trivial:

Insgesamt läuft der Auswertungsprozeß in drei Schritten ab:

1. Auffinden des nächsten Redex;
2. Reduzieren des Redex;
3. Aktualisierung des zum Redex gehörenden Wurzelknotens.

Dieser Prozeß wird solange wiederholt, bis sich kein Redex mehr finden läßt.

Grundsätzlich lassen sich zwei Arten von Redex unterscheiden: Eingebaute Funktionen und benutzerdefinierte Superkombinatoren. Handelt es sich bei der äußersten Funktionsanwendung um einen Superkombinator, so kann dieser grundsätzlich reduziert werden. Handelt es sich hingegen um eine eingebaute Funktion, so liegt nur dann ein Redex vor, wenn alle Funktionsargumente nicht weiter reduziert werden können (mit Ausnahme der einfachen Selektion, die, wie wir später sehen werden, eine Sonderbehandlung erfährt).

Das Auffinden der am weitesten links außen stehenden Funktionsapplikation ist relativ einfach: Beginnend beim Wurzelknoten folgt man dem linken Ast der Applikationsknoten, bis man auf den Namen eines Superkombinators, oder den einer eingebauten Funktion stößt. Die Kette der dabei durchlaufenen Applikationsknoten nennt man auch das Rückgrad (engl. *spine*) des Ausdrucks.

Ist der Name der äußersten Funktion ermittelt, bestimmt man, wie viele Argumente diese erwartet und geht im Graph entsprechend viele Applikationsknoten zurück – damit hat man den Wurzelknoten des nächsten potentiellen Redex gefunden. Steht keine ausreichende Zahl an Argumenten bereit, handelt es sich um eine partielle Applikation. Der zu berechnende Term liegt dann in Kopfnormalform (engl. *Weak Head Normal Form*) vor und selbst dann, wenn es unter den Funktionsargumenten weitere Redexe geben sollte, wird die Berechnung abgebrochen.

Ähnlich wie bei der Typberechnung, kann man die äußerste Funktionsanwendung durch den Einsatz eines Stacks ermitteln. Im Unterschied zur Typberechnung liegen allerdings keine ausgewerteten Ausdrücke, sondern Zeiger (bzw. Terme) auf dem Stack.

Nach dieser anschaulichen Betrachtung der Graphreduktion wollen wir im nächsten Abschnitt eine abstrakte Maschine zur Graphreduktion beschreiben.

10.2 Die *Template Instantiation Machine* (TIM)

Die *Template Instantiation Machine* (TIM)[2] bildet die Grundlage unseres *staged interpreters* für CoreEML. Zur besseren Einordnung und Abgrenzung wollen wir die wesentlichen Aspekte der TIM in diesem Abschnitt kurz vorstellen. Dabei beschränken wir uns auf eine vereinfachte Variante von CoreEML, ohne strikte Argumentauswertung und `Let`-Ausdrücke.

Es sei angemerkt, daß die TIM nicht zu den effizientesten virtuellen Maschinen zur Übersetzung von funktionalen Sprachen mit verzögerter Auswertung zählt. Moderne Übersetzer, wie z.B. der Glasgow Haskell Compiler, erzeugen Programmcode für virtuelle Maschinen, die der von Neumann Architektur besser angepaßt sind (z.B. Spineless Tagless G-Machine [135]). Wir haben uns hier bewußt für die TIM als Modell entschieden, da der ihr zu Grunde liegende Interpreter recht einfach konzipiert ist. Wir wollen sehen, wie weit entfernt wir von einem spezialisierten Ansatz sind, wenn wir unser Konzept der Übersetzung durch strukturelle Typanalyse anwenden.

Die TIM ist ein Interpreter, dessen Zustand aus drei Komponenten besteht:

- Einem *Heap*, in dem der Reduktionsgraph abgelegt ist;
- einem Stack zur Aufnahme von Heapadressen; und
- einer Funktion f, die Variablennamen auf Heapadressen abbildet.

Ein *Heap* ist, ähnlich wie ein Freispeicher, eine partielle Abbildung von Speicheradressen in CoreEML-Terme – genauer gesagt, in erweiterte CoreEML-Terme, wie wir gleich sehen werden.

Mit der Notation $\nu[a \mapsto t]$ beschreiben wir einen Heap ν, der an der Adresse a den Wert t speichert; mit $\nu(a)$ $(a \in \mathit{domain}(\nu))$ kann der an der Adresse a gespeicherte Wert erfragt werden und mit $\texttt{new}(\nu, t)$ erhält man einen Heap $\nu'[a \mapsto t]$ mit $a \notin \mathit{domain}(\nu)$.

CoreEML-Terme werden als verzeigerte Strukturen im Heap abgelegt; d.h. die Komponenten eines Applikationsknotens bestehen nicht aus Termen, sondern aus Heapadressen. Superkombinatordefinitionen werden ebenfalls im Heap abgelegt. Der Einfachheit halber stellen wir sie als $\texttt{SC}(\{\!|\mathit{args}|\!\}, t)$ dar, wobei args die Liste der Superkombinatorelemente und t die rechte Seite der Superkombinatordefinition ist. Gemeinsam genutzte Teilgraphen realisieren wir über Indirektionsknoten. Der Knoten $\texttt{Ind}(a)$ ist ein Verweis auf den Term, der im Heap an Adresse a abgelegt ist.

Zwei Reduktionsschritte sind für die TIM essentiell. Findet die TIM auf der Stackspitze die Adresse eines Applikationsknotens, so wird diese durch die Adressen der Komponenten dieses Knotens ersetzt:

$$\begin{array}{rl} & \{\!|a : s|\!\} \quad \nu[a : \mathit{App}(a_1, a_2)] \quad f \\ \Longrightarrow & \{\!|a_1 : a_2 : s|\!\} \quad \nu \quad f \end{array}$$

[2] In der Literatur wird das Kürzel TIM oft für die *Three Instruction Machine* verwendet. Diese hat jedoch nichts mit der *Template Instantiation Machine* zu tun.

Diese Regel wird solange angewendet, bis die Stackspitze keinen Verweis mehr auf einen Applikationsknoten enthält und entspricht damit dem „Abwickeln des Rückgrates" eines Ausdrucks auf dem Stack.

Referenziert der Knoten auf der Stackspitze eine Superkombinatordefinition, so wird deren rechte Seite instanziiert.

$$\begin{array}{ll} & \{\!|a_0 : a_1 : \cdots : a_n : s|\!\}\ \ \nu[a_0 : \mathtt{SC}(\{\!|x_1 : \cdots : x_n|\!\}, t) \\ \Longrightarrow & \{\!|a_r : s|\!\}\ \ \nu'[a_n : \mathtt{Ind}(a_r)\ \ f \\ \text{mit} & (\nu', a_r) = \mathit{instantiate}\ t\ \nu\ f[x_1 \mapsto a_1, \cdots, x_n \mapsto a_n] \end{array}$$

Die Funktion *instantiate* bekommt hierzu die rechte Seite der Superkombinatordefiniton t, den Heap μ und die Funktion f, erweitert um die Zuordnung der Parameter zu den Heapadressen, die als Argumente auf dem Stack liegen, übergeben. Als Ergebnis liefert sie einen erweiterten Heap ν', der die instanziierte rechte Seite enthält, sowie die Adresse a_r, die auf den Wurzelknoten des instanziierten Terms zeigt.

Die Adresse des Superkombinators und die Adressen der Argumente werden vom Stack entfernt und durch die Adresse des gerade instanziierten Superkombinators ersetzt. Im Heap wird der Wurzelknoten a_n des Superkombinatorredexes durch einen Indirektionsknoten ersetzt, der auf die instanziierte rechte Seite zeigt. Durch dieses *update* wird verhindert, daß ein Superkombinatorredex eventuell mehrfach ausgewertet wird.

Wir verzichten an dieser Stelle auf die Diskussion der Behandlung von eingebauten Operationen und verweisen den Leser auf die Darstellung von Peyton Jones und Lester [140, Seite 51 ff.].

10.3 Die *staged*-TIM

Die *staged*-TIM wirkt wie ein Übersetzer von CoreEML nach System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ und dient uns als Modell der Übersetzung von CoreEML nach C++ .

Im Unterschied zur TIM werden Terme nicht direkt ausgewertet, sondern eine System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$-Funktion generiert, die deren Auswertung übernimmt. Stack und Heap der TIM werden dadurch zu Laufzeitkomponenten, wobei wir den Stack als Polytypliste darstellen und zur Umsetzung des Heap auf die in Abschnitt 2.7.3 eingeführten imperativen Sprachmerkmale zurückgreifen.

Der Zustand der *staged*-TIM besteht aus drei Komponenten:

- Einem Stack, auf dem CoreEML-Terme in Typdarstellung abgelegt werden;
- einer Aufrufspur Δ, in der die in Übersetzung befindlichen Superkombinatorinstanzen vermerkt sind; und
- einer Funktion Θ, die Laufzeitstacks manipuliert und in der das Ergebnis der Übersetzung akkumuliert wird.

Bei der TIM wird die Wahl einer Reduktionsregel allein von der Struktur des Stacks bzw. der darauf befindlichen Terme gesteuert. Da CoreEML-Terme und deren Typen strukturäquivalent sind, können wir die *staged*-TIM also durch einen Stack steuern, auf dem wir Typen ablegen.

staged-TIM I

$$\frac{\{\!|\alpha_f : \alpha_a|\!\} \;\; \Delta \;\; \mathtt{id} \rightarrow \{\!|T_r|\!\} \;\; \Delta \;\; \Theta_1}{\{\!|\mathtt{App}(\alpha_f, \alpha_a) : \alpha_s|\!\} \;\; \Delta \;\; \Theta \rightarrow \{\!|T_r : \alpha_s|\!\} \;\; \Delta \;\; \Theta_1 \circ (\mathtt{f}_{App} \circ \Theta)} \quad \textbf{(ST-App)}$$

mit $\mathtt{f}_{App} := \lambda s : \{\!|\mathtt{App}(\alpha_f, \alpha_a) : \alpha : s|\!\}$.
$\quad \mathtt{case}\ s\ \mathtt{of}$
$\quad \langle \{\!|\mathtt{App}(\alpha_f, \alpha_a) : \alpha : s|\!\} \ ; \ \{\!|\mathtt{App}(f, a) : s|\!\} \rangle \Rightarrow \{\!|f : a : s|\!\}$

$$\frac{\{\!|\alpha_1|\!\} \;\; \Delta \;\; \mathtt{id} \rightarrow \{\!|T_1|\!\} \;\; \Delta \;\; \Theta_1 \quad \{\!|\alpha_2|\!\} \;\; \Delta \;\; \mathtt{id} \rightarrow \{\!|T_2|\!\} \;\; \Delta \;\; \Theta_2}{\begin{array}{c}\{\!|\mathtt{OpPlus} : \alpha_1 : \alpha_2 : \alpha_s|\!\} \;\; \Delta \;\; \Theta \\ \rightarrow \quad \{\!|\mathtt{bop}_{\mathbb{T}}[\mathtt{OpPlus} \times T_1 \times T_2] : \alpha_s|\!\} \;\; \Delta \;\; \mathtt{f}_{\mathtt{OpPlus}} \circ \Theta\end{array}} \quad \textbf{(ST-OpPlus)}$$

mit $\mathtt{f}_{\mathtt{OpPlus}} := \lambda s : \{\!|\mathtt{OpPlus} : \alpha_1 : \alpha_2 : \alpha_s|\!\}$.
$\quad \mathtt{case}\ s\ \mathtt{of}$
$\quad \langle \{\!|\mathtt{OpPlus} : \alpha_1 : \alpha_2 : \alpha_s|\!\} \ ; \ \{\!|v : x_1 : x_2 : s|\!\} \rangle \Rightarrow$
$\quad\quad \mathtt{let}\ a_1 = (\Theta_1 \ \{\!|x_1|\!\}) .0$
$\quad\quad\quad\ a_2 = (\Theta_2 \ \{\!|x_2|\!\}) .0$
$\quad\quad\quad\ r = \mathtt{bop}_{\mathbb{V}}[\mathtt{OpPlus} \times T_1 \times T_2](a_1, a_2)$
$\quad\quad\quad\ \mathtt{in}\ \{\!|r : s|\!\}$

$$\frac{\begin{array}{c}\{\!|\alpha_c|\!\} \;\; \Delta \;\; \mathtt{id} \rightarrow \{\!|\mathtt{bool}|\!\} \;\; \Delta \; \Theta_c \quad \{\!|\alpha_t|\!\} \;\; \Delta \;\; \mathtt{id} \rightarrow \{\!|T_t|\!\} \;\; \Delta \;\; \Theta_t \\ \{\!|\alpha_r|\!\} \;\; \Delta \;\; \mathtt{id} \rightarrow \{\!|T_e|\!\} \;\; \Delta \;\; \Theta_e \\ T_r = \mathtt{If}_{\mathbb{T}}[T_c \times T_t \times T_e]\end{array}}{\{\!|\mathtt{OpIf} : \alpha_c : \alpha_t : \alpha_e : \alpha_s|\!\} \;\; \Delta \;\; \Theta \rightarrow \{\!|\mathtt{if}_{\mathbb{T}}[\alpha_c \times \alpha_t \times \alpha_e] : \alpha_s|\!\} \;\; \Delta \;\; \mathtt{f}_{\mathtt{OpIf}} \circ \Theta} \quad \textbf{(ST-OpIf)}$$

mit $\mathtt{f}_{\mathtt{OpIf}} := \lambda s : \{\!|\mathtt{OpIf} : \alpha_c : \alpha_t : \alpha_e : \alpha_s|\!\}$.
$\quad \mathtt{case}\ s\ \mathtt{of}$
$\quad \langle \{\!|\mathtt{OpIf} : \alpha_c : \alpha_t : \alpha_e : \alpha_s|\!\} \ ; \ \{\!|v : x_c : x_t : x_e : s|\!\} \rangle \Rightarrow$
$\quad\quad \mathtt{let}\ a_c = (\Theta_c \, \{\!|x_c|\!\}).0$
$\quad\quad\quad\ \mathtt{in}\ \mathtt{if}\ a_c = \mathtt{true}\ \mathtt{then}\ \{\!|(\Theta_t \, \{\!|x_t|\!\}) .0 : s|\!\}$
$\quad\quad\quad\quad\ \mathtt{else}\ \{\!|(\Theta_e \, \{\!|x_e|\!\}) .0 : s|\!\}$

Abbildung 10.1: Transitionsregeln der *staged*-TIM für Grundoperationen.

Befindet sich beispielsweise ein Applikationsknotentyp $\mathtt{App}(\alpha_f, \alpha_a)$ auf dem Stack, so versetzen wir die *staged*-TIM in den Zustand $\alpha_f : \alpha_a$ und lassen sie eine Funktion Θ_1 generieren, die Werte von diesem Stacktyp übernimmt und daraus einen Stack generiert, der ausschließlich einen Wert vom Typ T_r speichert.

Die Übersetzungsfunktion wird durch Funktionskomposition gebildet, wobei wir immer mit der polymorphen Identitätsfunktion `id` starten[3].

Ist der Programmcode zur Abarbeitung des aufgespaltenen Applikationsknotens verfügbar, können wir Code für den Applikationsknoten selbst erzeugen. Die Funktion $\mathtt{f}_{\mathtt{App}}$ übernimmt den aktuellen Stack als Argument, zerlegt ihn mittels `case` in seine Bestandteile und generiert einen Stack, auf dem Funktions- und Argumentteil abgelegt sind. Das Gesamtergebnis der Übersetzung ergibt sich nun durch Funktionskomposition aus Θ_1, $\mathtt{f}_{\mathtt{App}}$ und der bisher generierten Übersetzungsfunktion Θ (siehe Abbildung 10.1 – Regel **ST-App**).

[3]Streng genommen müßten wir `id` natürlich noch mit dem entsprechenden Stacktyp initialisieren.

Wie bei der TIM, wird die Regel **ST-App** solange ausgeführt, bis sich kein Applikationsknotentyp mehr auf der Stackspitze befindet.

Liegt der Typ eines eingebauten Operationssymbols oben auf dem Stack, so müssen die Argumente strikt ausgewertet werden. Im Fall von `OpPlus` (siehe Regel **ST-OpPlus**) werden daher zunächst die Funktionen Θ_1 und Θ_2 zur Auswertung der Argumente vom Typ α_1 und α_2 generiert. Diese liefern im Ergebnis Stacks, die allein einen Wert vom Typ T_1 bzw. T_2 enthalten.

Der Interpreter wechselt dann in den Zustand

$$\{\!|\mathtt{bop}_{\mathbb{T}}[\mathtt{OpPlus} \times T_1 \times T_2] : \alpha_s|\!\} \quad \Delta \ \mathtt{f}_{\mathtt{OpPlus}} \circ \Theta$$

Auf dem Typstack wurden Operationssymbol und Argumenttypen durch den Ergebnistyp der Addition ersetzt und die Auswertungsfunktion Θ wurde um eine Funktion $\mathtt{f}_{\mathtt{OpPlus}}$ erweitert, die die eigentliche Addition vollzieht. Sie erhält einen Stack vom Typ $\{\!|\mathtt{OpPlus} : \alpha_1 : \alpha_2 : \alpha_s|\!\}$ als Eingabe und zerlegt diesen mittels `case`-Analyse in seine Bestandteile.

In einem `let`-Ausdruck wird den Variablen a_1 und a_2 das Ergebnis der Auswertung der Argumente x_1 und x_2 zugewiesen. Diese ergeben sich durch Anwendung der Funktionen Θ_1 und Θ_2 auf einen Stack, der lediglich das entsprechende Argument (x_1 bzw. x_2) enthält. Das Ergebnis der Auswertung liegt jeweils auf dem als Ergebnis gelieferten Stack und kann durch Projektion auf die nullte Komponente selektiert werden.

In der lokalen Variable r wird das Ergebnis der Additionsoperation zwischengespeichert. Die Additionsfunktion selbst wird durch Anwendung der Codeselektionsfunktion $\mathtt{bop}_{\mathbb{V}}$ auf $\mathtt{OpPlus} \times T_1 \times T_2$, also einen Tupeltyp, dessen zweite und dritte Komponente den Typen von a_1 und a_2 entspricht, gewählt und auf das Paar (a_1, a_2) angewendet. Als Ergebnis generiert $\mathtt{f}_{\mathtt{OpPlus}}$ einen Stack, dessen Spitze den Wert r enthält.

Die Übersetzung von `if` verläuft nahezu analog (siehe Regel **ST-If**). Es wird Programmcode zur Auswertung der Bedingung (Θ_c), dem then- (Θ_t) und dem else-Zweig (Θ_e) erzeugt, um darauf aufbauend die Funktion $\mathtt{f}_{\mathtt{OpIf}}$ zu generieren.

Interessanter ist die Übersetzung von Superkombinatorapplikationen. Im Vergleich zur TIM liegen keine Heapadressen, sondern CoreEML-Terme auf dem Stack. Abhängig davon, wie der jeweilige Superkombinator einzelne Argumente auswertet (strikt oder nicht-strikt), werden diese entweder ausgewertet, oder aber auf den Heap verschoben.

Die Funktionen $\mathtt{mkArg}_{\mathbb{T}}$ und $\mathtt{mkArg}_{\mathbb{V}}$ übernehmen diese Behandlung der Argumente (siehe Abbildung 10.2). $\mathtt{mkArg}_{\mathbb{T}}$ berechnet den Typ des angepaßten Arguments. Sie erhält drei Typen als Eingabe: α_{Para} übernimmt die Typdarstellung des Parameters. Dabei handelt es sich entweder um eine Variable, also einen Typ $\mathtt{Var}(\alpha)$, oder aber um einen Typ $\mathtt{App}(\mathtt{OpNot}, \mathtt{Var}(\alpha))$, falls der Parameter mit einer Striktheitsannotation versehen wurde. In α_{Arg} wird der Typ des Arguments an den Superkombinator übergeben. Der Parameter α_{Res} übernimmt schließlich den Typ des Wertes, der sich durch Auswertung eines CoreEML-Terms vom Typ α_{Arg} ergibt.

Wird ein Applikationsknoten an einen Parameter gebunden, der nicht-strikt auszuwerten ist, wird ein Heapknoten erzeugt. Auf dem Heap werden im Vergleich zur TIM jedoch keine Terme, sondern Funktionsabschlüsse gespeichert, die wir als Funktionen vom Typ $\mathtt{Unit} \rightarrow \alpha_{Res}$ darstellen. Alle anderen Knotentypen bleiben unangetastet.

Ist eine Funktion in einem bestimmten Argument strikt, wird das Argument zu einer Normalform reduziert, bevor es in den Rumpf der Superkombinatordefinition substituiert wird.

$\texttt{mkArg}_{\mathbb{T}}$ mkArg

$$
\begin{array}{l}
\texttt{mkArg}_{\mathbb{T}} := \Lambda\alpha_{Para} :: *.\Lambda\alpha_{Arg} :: *.\Lambda\alpha_{Res} :: *. \\
\quad \texttt{Match}\ \alpha_{Para}\ \texttt{with} \\
\quad\quad \texttt{Var}(\alpha) \Rightarrow \texttt{Match}\ \alpha_{Arg}\ \texttt{with} \\
\quad\quad\quad\quad \texttt{App}(\alpha_F, \alpha_A) \Rightarrow \texttt{Ref}(\texttt{Unit} \rightarrow \alpha_{Res}) \\
\quad\quad\quad\quad \texttt{else}\ \Lambda\delta :: *.\delta \\
\quad\quad \texttt{App}(\texttt{OpNot}, \texttt{Var}(\alpha)) \Rightarrow \alpha_{Result}
\end{array}
$$

$\texttt{mkArg}_{\mathbb{V}}$

$$
\begin{array}{l}
\texttt{mkArg}_{\mathbb{V}} := \Lambda\alpha_{Para} :: *.\Lambda\alpha_{Arg} :: *.\Lambda\alpha_{Res} :: *.\lambda f : \{\!|\alpha_{Arg}|\!\} \rightarrow \{\!|\alpha_{Res}|\!\}\,. \\
\quad \texttt{match}\ \alpha_{Para}\ \texttt{with} \\
\quad\quad \texttt{Var}(\alpha) \Rightarrow \texttt{match}\ \alpha_{Arg}\ \texttt{with} \\
\quad\quad\quad\quad \texttt{App}(\alpha_f, \alpha_a) \Rightarrow \lambda x : \texttt{App}(\alpha_f, \alpha_a). \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{new}(\lambda z : \texttt{Unit}.f\ \{\!|x|\!\}) \\
\quad\quad\quad\quad \texttt{else}\ \Lambda\delta :: *.\lambda x : \delta.x \\
\quad\quad \texttt{App}(\texttt{OpNot}, \texttt{Var}(\alpha)) \Rightarrow \lambda x : \alpha_{Arg}.f\ \{\!|x|\!\}
\end{array}
$$

Abbildung 10.2: Funktionen zur Behandlung der Argumente an einen Superkombinator.

Die Funktion $\texttt{mkArg}_{\mathbb{V}}$ generiert eine Funktion zur Anpassung der Argumentwerte. Sie erwartet im Vergleich zur Typfunktion $\texttt{mkArg}_{\mathbb{T}}$ als zusätzliches Argument eine Funktion f vom Typ $\{\!|\alpha_{Arg}|\!\} \rightarrow \{\!|\alpha_{Res}|\!\}$, die von der *staged*-TIM zur Reduktion von Argumenten mit Typ α_{Arg} generiert wurde.

Wenn ein Applikationsknoten x an einen nicht-strikten Parameter gebunden wird, wird eine Funktion $\lambda z : \texttt{Unit}.f\ \{\!|x|\!\}$ auf dem Heap abgelegt und eine entsprechende Referenz zurückgegeben. Durch die λ-Abstraktion (den Parameter z) erreichen wir, daß die Anwendung von f auf den Stack $\{\!|x|\!\}$ zurückgestellt wird, bis wir mit dem Wert `unit` applizieren.

Bei strikter Argumentauswertung wird f unmittelbar auf $\{\!|x|\!\}$ angewendet, sofern ein Applikationsknoten übergeben wurde. Ansonsten bleibt das Argument unverändert.

Die mit $\texttt{mkArg}_{\mathbb{V}}$ behandelten Argumente werden später in die rechte Seite einer Superkombinatordefinition eingesetzt. Wir müssen daher festlegen, wie bei der Auswertung von Referenzen auf Funktionsabschlüsse verfahren werden soll (siehe Abbildung 10.3, Regel **ST-Ref**).

Trifft der Interpreter auf einen Stack, dessen Spitze eine Referenz vom Typ $\texttt{Ref}(\texttt{Unit} \rightarrow \alpha_r)$ enthält, so ist der Funktionsabschluß auszuwerten, wobei ein Wert vom Typ α_r entsteht. Die Funktion $\texttt{f}_{\texttt{Ref}}$ übernimmt einen entsprechenden Stack und zerlegt ihn, wie üblich, mittels *case*-Analyse in seine Bestandteile. Dabei wird die Variable x an die auf der Stackspitze befindliche Referenz gebunden.

In einem `let`-Ausdruck wird der Variablen a das Ergebnis des Funktionsabschlusses, auf den x zeigt, zugewiesen. Dazu wird x dereferenziert und die dabei gewonnene Funktion auf die Konstante `unit` angewendet. Die Essenz der verzögerten Auswertung (der *Update* von Knoten) wird durch die sich anschließende Zuweisung erreicht, indem die Heapspeicherzelle x mit einer neuen Funktion überschrieben wird, die bei Anwendung auf `unit` sofort zum Ergebnis a führt.

Als Ergebnis liefert $\texttt{f}_{\texttt{Ref}}$ ein Stack, auf dessen Spitze das ausgewertete Argument a liegt.

Bei der Auswertung von Superkombinatordefinitionen kommt die Menge Δ ins Spiel. Liegen ein Superkombinatorname und ausreichend viele Argumente auf dem Stack, um ihn zu instanziie-

staged-Tim II

$$\{\!|\texttt{Ref}(\texttt{Unit} \to \alpha_r) : \alpha_s|\!\}\ \ \Delta\ \ \Theta \to \{\!|\alpha_r : \alpha_s|\!\}\ \ \Delta\ \ \mathtt{f}_{\texttt{Ref}} \circ \Theta \quad \textbf{(ST-Ref)}$$

mit $\mathtt{f}_{\texttt{Ref}} := \lambda s : \{\!|\texttt{Ref}(\texttt{Unit} \to \alpha_r) : \alpha_s|\!\}\,.$

```
case s of
⟨{|Ref(Unit → α_r) : α_s|} ; {|x : s|}⟩ ⇒
    let a = deref(x) unit
        in x := λz : Unit.a;
            {|a : s|}
```

$$\frac{\begin{array}{c} v(t_1)\cdots(t_n) = t \in \mathcal{P}_{EML} \\ \{\!|\alpha_1|\!\}\ \ \Delta\ \ \mathtt{id} \to \{\!|T_{a1}|\!\}\ \ \Delta\ \ \Theta_1\ \ \cdots\ \ \{\!|\alpha_n|\!\}\ \ \Delta\ \ \mathtt{id} \to \{\!|T_{an}|\!\}\ \ \Delta\ \ \Theta_n \\ T_{b1} = \mathtt{mkArg}_{\mathbb{T}}[T_1][\alpha_1][T_{a1}] \\ \cdots \\ T_{bn} = \mathtt{mkArg}_{\mathbb{T}}[T_n][\alpha_n][T_{an}] \\ v(T_{b1})\cdots(T_{bn}) \notin \Delta \\ \{\!|T[T_{b1}/\alpha_1, \cdots, T_{bn}/\alpha_n]|\!\}\ \ \Delta \cup \{v(T_{b1})\cdots(T_{bn})\}\ \ \mathtt{id} \to \{\!|T_r|\!\}\ \ \Delta'\ \ \Theta_r \end{array}}{\{\!|\texttt{Var}(\alpha) : \alpha_1 : \cdots : \alpha_n : \alpha_s|\!\}\ \ \Delta\ \ \Theta \to \{\!|T_r : \alpha_s|\!\}\ \ \Delta\ \ \mathtt{f}_{SC} \circ \Theta} \quad \textbf{(ST-SC)}$$

mit $\mathtt{f}_{SC} := \lambda s : \{\!|\texttt{Var}(\alpha) : \alpha_1 : \cdots : \alpha_n : \alpha_s|\!\}\,.$

```
case s of
⟨{|Var(α) : α_1 : ··· : α_n : α_s|} ; {|v : x_1 : ··· : x_n : s|}⟩ ⇒
    let b_1 = mkArg_𝕍[T_1][α_1][T_a1] Θ_1 x_1
        ⋮
        b_n = mkArg_𝕍[T_n][α_n][T_an] Θ_n x_n
        in (fix v_{T_b1,···,T_bn} : {|T[T_b1/α_1 : ··· : T_bn/α_n|} → {|T_r|}.Θ_r)
            (instantiate t {|b_1 : ··· : b_n|})
```

$$\frac{\begin{array}{c} v(t_1)\cdots(t_n) = t \in \mathcal{P}_{EML} \\ \{\!|\alpha_1|\!\}\ \ \Delta\ \ \mathtt{id} \to \{\!|T_{a1}|\!\}\ \ \Delta\ \ \Theta_1\ \ \cdots\ \ \{\!|\alpha_n|\!\}\ \ \Delta\mathtt{id} \to \{\!|T_{an}|\!\}\ \ \Delta\ \ \Theta_n \\ T_{b1} = \mathtt{mkArg}_{\mathbb{T}}[T_1][\alpha_1][T_{a1}] \\ \cdots \\ T_{bn} = \mathtt{mkArg}_{\mathbb{T}}[T_n][\alpha_n][T_{an}] \\ v(T_{b1})\cdots(T_{bn}) \in \Delta \\ \{\!|T[T_{a1}/\alpha_1, \cdots, T_{an}/\alpha_n]|\!\}\ \ \emptyset \Downarrow_{\mathbb{T}} \{\!|T_r|\!\}\ \ ct \end{array}}{\{\!|\texttt{Var}(\alpha) : \alpha_1 : \cdots : \alpha_n : \alpha_s|\!\}\ \ \Delta\ \ \Theta \to \{\!|T_r : \alpha_s|\!\}\ \ \Delta\ \ \mathtt{f}_{SCRek} \circ \Theta} \quad \textbf{(ST-SCRek)}$$

mit $\mathtt{f}_{SCRek} := \lambda s : \{\!|\texttt{Var}(\alpha) : \alpha_1 : \cdots : \alpha_n : \alpha_s|\!\}\,.$

```
case s of
⟨{|Var(α) : α_1 : ··· : α_n : α_s|} ; {|v : x_1 : ··· : x_n : s|}⟩ ⇒
    let b_1 = mkArg_𝕍[T_1][α_1][T_a1] Θ_1 x_1
        ⋮
        b_n = mkArg_𝕍[T_n][α_n][T_an] Θ_n x_n
        in v_{T_b1,···,T_bn}(instantiate t {|b_1 : ··· : b_n|})
```

Abbildung 10.3: Übersetzung von Superkombinatoren und Referenzen.

ren, so werden die Argumente mit $\texttt{mkArg}_{\mathbb{V}}$ entsprechend der Auswertungsstrategie angepaßt. Anschließend wird der Superkombinator mit den so behandelten Argumenten instanziiert und auf dem Stack abgelegt, um dann eine Funktion zur Auswertung der instanziierten rechten Seite zu generieren.

Ist ein Superkombinator rekursiv definiert, würde die *staged*-TIM in einer Endlosschleife immer wieder Code zur Auswertung derselben Superkombinatorapplikation erzeugen. Hier ein Beispiel:

```
( fac(!x) = If(x==0, 1, x*fac(x-1) )
)[ fac(3) ]
```

Initial startet der *staged interpreter* im Zustand

$$\{|\texttt{App}(\texttt{fac}, \texttt{int})|\} \quad \emptyset \quad \texttt{id}$$

und erreicht durch Anwenden der Regel **ST-App** den Zustand

$$\{|\texttt{fac} : \texttt{int}|\} \quad \emptyset \quad \Theta$$

Es liegt ein Superkombinator mit ausreichend vielen Argumenten auf dem Typstack. Das Argument `int` bleibt von $\texttt{mkArg}_{\mathbb{V}}$ unberührt, so daß wir den Superkombinator direkt instanziieren und auf dem Stack ablegen können:

$$\left\{\left|\texttt{App}(\texttt{App}(\texttt{App}(\texttt{OpIf}, \cdots), \texttt{int}), \texttt{App}(\cdots, \texttt{App}(\texttt{fac}, \texttt{App}(\texttt{OpSub}, \texttt{int}), \texttt{int}))\right|\right\}$$

Interessant ist hier der rekursive Aufruf von `fac`. Nach weiteren Reduktionsschritten kommt die *staged* TIM in den Zustand

$$\{|\texttt{fac} : \texttt{App}(\texttt{App}(\texttt{OpSub}, \texttt{int}), \texttt{int})|\} \quad \Delta \quad \Theta$$

Da der Parameter x mit einer Striktheitsannotation versehen wurde, wird der Typ

$$\texttt{App}(\texttt{App}(\texttt{OpSub}, \texttt{int}), \texttt{int})$$

von $\texttt{mkArg}_{\mathbb{T}}$

zu `int` reduziert und die Fakultätsfunktion abermals mit `int` instanziiert und auf dem Stack abgelegt – der Interpreter dreht sich im Kreis.

Um dies zu vermeiden, führen wir in Δ Buch darüber, für welche Superkombinatoraufrufe gerade Programmcode generiert wird und unterscheiden initiale und rekursive Aufrufe.

Initiale Aufrufe werden von der Regel **ST-SC** behandelt. Voraussetzung ist, daß das EML-Programm $\mathcal{P}_{EML}$ einen Superkombinator v bereithält. Die Typdarstellung des zugehörigen CoreEML-Terms haben wir nicht explizit angegeben. Wir setzen voraus, daß die Parameter t_1 bis t_n vom Typ T_1 bis T_n und die rechte Seite t vom Typ T ist.

Zunächst werden Funktionen Θ_1 bis Θ_n generiert, um Argumente vom Typ α_1 bis α_n zu Normalformen vom Typ T_{a_1} bis T_{a_n} zu reduzieren. Anschließend wird berechnet, in welcher Form die Argumente in die rechte Seite der Superkombinatordefinition einzusetzen sind. Dazu

wird die Funktion $\texttt{mkArg}_{\mathbb{T}}$ jeweils mit dem Parametertyp (T_1 bis T_n), dem Argumenttyp (α_1 bis α_n) und dem Typ der zugehörigen Normalform (T_{a_1} bis T_{a_n}) aufgerufen.

Es folgt der Test, ob es sich um einen rekursiven Aufruf handelt. Dies ist der Fall, wenn die Aufrufspur Δ den CoreEML-Term $v(\alpha_1)\cdots(\alpha_n)$ enthält.

In der letzten Prämisse wird eine Funktion Θ_r zur Auswertung des instanziierten Superkombinators erzeugt. Dazu wird der Interpreter mit einem Stack gestartet, der ausschließlich den instanziieren Superkombinatortyp enthält, und einer Aufrufspur, die um den Typterm $v(\alpha_1)\cdots(\alpha_n)$ erweitert ist.

Die eigentliche Berechnung wird von der Funktion $\texttt{f}_{SC}$ durchgeführt. Sie erhält einen entsprechenden Stack als Eingabe, der mittels *case*-Analyse in seine Bestandteile zerlegt wird. Durch einen `let`-Ausdruck werden den Variablen b_1 bis b_n die gewandelten Argumente zugeordnet. Um rekursive Superkombinatoren zu erlauben, starten wir mit einer Fixpunktberechnung. $v_{T_{b_1},\cdots,T_{b_n}}$ ist eine von der *staged*-TIM frisch generierte Funktionsvariable und steht für eine Funktion, die einen Stack, der den instanziierten Superkombinator enthält, auf einen Stack abbildet, der einen Wert vom Ergebnistyp T_r speichert. Der Fixpunktoperator wird auf die Funktion Θ_r angewendet, die zur Auswertung des instanziierten Superkombinators generiert wurde. Die durch Fixpunktiteration gewonnene Funktion wird schließlich auf einen Stack angewendet, der den instanziierten Superkombinator enthält.

Zur Berechnung von Θ_r wurde die Aufrufspur Δ um den Eintrag $v(T_{b_1})\cdots(T_{b_n})$ erweitert. Enthält die rechte Seite einen rekursiven Aufruf, so wird dieser mit der Regel **ST-SCRek** behandelt. Anstatt erneut eine Funktion Θ_r zu generieren, wird die in der Fixpunktberechnung eingeführte Funktionsvariable $v_{T_{b_1},\cdots,T_{b_n}}$ auf den instanziierten Superkombinator angewendet (siehe Regel **ST-SCRek**).

Abbildung 10.4 zeigt die Regel **ST-Let** zur Übersetzung von `let`-Ausdrücken.

10.4 Implementierung der *staged*-TIM

Die im vorigen Abschnitt vorgestellte *staged*-TIM läßt sich recht einfach in C++ Template-Metaprogramme übersetzen, indem man die in Kapitel 5 vorgestellten Analogien zwischen System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ und C++ berücksichtigt.

Etwas einfacher wird die Übersetzung von Superkombinatoraufrufen. Da der Template Mechanismus für jedes instanziierte Template automatisch einen neuen Namen generiert, kann man auf die Kodierung mittels Fixpunktkombinator verzichten und muß nicht Buch über die rekursiven Aufrufe führen.

In C++ kann man die *staged*-TIM über eine Metafunktion `translateCorelEML` realisieren, die neben dem CoreEML-Programm in Typdarstellung einen Typstack als Argument erhält und eine Funktion generiert, die einen Laufzeitstack als Eingabe erhält und daraus einen Wert berechnet.

Die Regel **ST-App** könnte z.B. wie folgt realisiert werden:

staged-TIM III

$$\frac{\begin{array}{c}\{\!|T_1|\!\}\ \ \Delta\ \ \mathtt{id} \rightarrow \{\!|T_{v_1}|\!\}\ \ \Delta\ \ \Theta_1 \\ \{\!|T_2[T_{v_1}/\mathtt{Var}(\alpha_1)]|\!\}\ \ \Delta\ \ \mathtt{id} \rightarrow \{\!|T_{v_2}|\!\}\ \ \Delta\ \ \Theta_2 \\ \vdots \\ \{\!|T_n[T_{v_1}/\mathtt{Var}(\alpha_1),\cdots,T_{v_{n-1}}/\mathtt{Var}(\alpha_{n-1})|\!\}\ \ \Delta\ \ \mathtt{id} \rightarrow \Delta\ \ \Theta_n \\ \{\!|T[T_{v_1}/\mathtt{Var}(\alpha_1),\cdots,T_{v_n}/\mathtt{Var}(\alpha_n)]|\!\}\ \ \Delta\ \ \mathtt{id} \rightarrow \{\!|T_r|\!\}\ \ \Delta\ \ \Theta_r\end{array}}{\begin{array}{l}\{\!|\mathtt{OpLet} : \mathtt{Let}(\{\!|\mathtt{Var}(\alpha_1) = T_1,\cdots,\mathtt{Var}(\alpha_n) = T_n|\!\},T) : \alpha_s|\!\}\ \ \Delta\ \ \Theta \\ \rightarrow \{\!|T_r : \alpha_s|\!\}\ \ \Delta\ \ \mathtt{f}_{\mathtt{OpLet}} \circ \Theta\end{array}}\ \textbf{(ST-Let)}$$

mit $\mathtt{f}_{\mathtt{OpLet}} :=$

$$\begin{array}{l}\lambda s : \{\!|\mathtt{OpLet} : \mathtt{Let}(\{\!|\mathtt{Var}(\alpha_1) = T_1,\cdots,\mathtt{Var}(\alpha_n) = T_n|\!\},T) : \alpha_s|\!\}\,. \\ \mathtt{case}\ s\ \mathtt{of} \\ \quad \langle \{\!|\mathtt{OpLet} : \mathtt{Let}(\{\!|\mathtt{Var}(\alpha_1) = T_1,\cdots,\mathtt{Var}(\alpha_n) = T_n|\!\},T) : \alpha_s|\!\}\ ; \\ \quad\ \ \{\!|v : \mathtt{Let}(\{\!|v_1 = t_1,\cdots,v_n = t_n|\!\},t) : s|\!\}\rangle \Rightarrow \\ \quad \mathtt{let}\ \ a_1 = (\Theta_1\ \ \{\!|t_1|\!\}).0 \\ \qquad\quad a_2 = (\Theta_2\ \ \{\!|t_2[a_1/v_1]|\!\}).0 \\ \qquad\quad \vdots \\ \qquad\quad a_n = (\Theta_n\ \ \{\!|t_n[a_1/v_1,\cdots,a_{n-1}/v_n{-}1]|\!\}).0 \\ \qquad\quad r = (\Theta_r\ \ \{\!|t[a_1/v_1,\ldots,a_n/v_n]|\!\}).0 \\ \qquad\ \ \mathtt{in}\ \ \{\!|r : s|\!\}\end{array}$$

Abbildung 10.4: Übersetzung von `let`-Ausdrücken.

```
using namespace TML;

template <typename SCRIPT,typename STACK>
struct translateCoreEML {};

template<typename SCRIPT,typename F,typename A,typename S>
struct translateCoreEML<SCRIPT,C<App<F,A>,S> > {
  typedef translateCoreEML<SCRIPT, C<F,C<A,S> > >::RET RET;

  static inline RET exec(const C<App<F,A>,S> >& stack) {
    return translateCoreEML<SCRIPT, C<F, C<A,S> >::exec(
             cons(stack.h.f, cons(stack.h.a, stack.t)) );
  }
};
```

Wie im Modell wird ein neuer Stack kreiert und an die entsprechende Auswertungsfunktion weitergereicht.

Zwei Punkte machen diese eins zu eins Umsetzung unattraktiv: Zum einen ist es nicht besonders günstig, komplette Stacks als Argumente zu übergeben bzw. bei jedem Neustart des Interpreters einen neuen Stack zu konstruieren. Zum anderen werden Objekte, die beim Instanziieren eines Superkombinators auf den Heap verschoben werden, niemals gelöscht, so daß bei längeren Berechnungen schnell der Speicherbedarf einer Applikation außer Kontrolle gerät.

In den folgenden Abschnitten werden wir darlegen, wie wir den Stack für CoreEML implementiert haben und wie man in C++ einen Heap so realisieren kann, daß Objekte automatisch freigegeben werden, wenn sie nicht mehr benötigt werden.

```
class EMLStack {
  char *theStack, *stackPtr;
public:
  EMLStack(int size) { theStack = new char[size];
                       stackPtr = theStack + size;
                     }
  ~EMLStack()        { delete [] theStack; }
  template <typename T>
  void push(const T& value) {
    stackPtr -= sizeof(T);
    new (stackPtr) T(value);
  }
  template <typename T>
  T pop() {
    T retVal = *(T*)stackPtr;
    reinterpret_cast<T*>(stackPtr)->~T();
    stackPtr += sizeof(T);
    return retVal;
  }
  template <typename T>
  T& top() const {
    return *(T*)stackPtr;
  }
};
```

Abbildung 10.5: Grundzüge der Implementierung des Stacks der *staged* TIM.

10.4.1 Implementierung des Stacks

Im Grunde genommen brauchen wir zur Umsetzung der Transitionen der TIM nur Zugriff auf die obersten n Stack-Elemente. Den Typ dieser Elemente können wir der Zustandsinformation der *staged*-TIM entnehmen.

Den Stack können wir daher als einfaches Byte-Feld darstellen, und über Typkonvertierungen auf die entsprechenden Elemente zugreifen. `push` und `pop` werden dadurch typabhängige Operationen.

Der Konstruktor `EMLStack` übernimmt die Kapazität des Stack in Byte, reserviert entsprechend Speicher und bewegt den Stackzeiger an das Ende dieses Speicherbereichs.

Im Destruktor `~EMLStack` wird der reservierte Speicherbereich wieder freigegeben.

Im Methodentemplate `push` nutzen wir den *placement new* Operator. Diesem zweistelligen Operator wird im ersten Argument die Adresse des zu erzeugenden Objekts übergeben, im zweiten Argument erhält er das Objekt selbst. Der Effekt ist, daß ein neues Objekt vom Typ `T` über einen Kopierkonstruktor an der angegebenen Speicherstelle kreiert wird.

Die Stackspeicherzelle über eine Zuweisung `*(T*)stackPtr = value` zu initialisieren könnte zu Fehlern führen, da der Zuweisungsoperator auf der linken Seite ein gültiges Objekt erwartet, der entsprechende Speicherbereich in diesem Fall jedoch uninitialisiert ist.

Beim Entfernen eines Objekts mittels `pop` muß im Gegenzug der Destruktor der Klasse `T` aufgerufen werden, um das Objekt auf dem Stack ordnungsgemäß zu deinitialisieren, bevor es

entfernt wird.

Der Wert von CoreEML-Variablen, die sich auf dem Stack befinden, wird in den erzeugten Funktionen nicht gebraucht. Um zu verhindern, daß diese unnötigerweise auf dem Stack abgelegt werden, können wir die Methodentemplates `push` und `pop` für Typen der Form `Var<int>` spezialisieren, so daß sie keine Operation ausführen.

Wir können aber noch weitere Optimierungen vornehmen. Die zentrale Aufgabe der Regel **ST-App** ist das Aufspalten eines `App`-Knotens. Der Funktionsteil wird auf die Stackspitze und unmittelbar dahinter der Applikationsteil gelegt. Sofern das Speicherlayout von `App`-Struktur und Stack übereinstimmt, brauchen wir keine Änderungen am Stack vornehmen und das Abwickeln des Rückgrates eines Terms könnte entfallen.

Damit diese Optimierung greift, dürfen Applikationsknoten keine Werte von CoreEML-Variablen speichern – diese legen wir ja nicht auf dem Stack ab. Wir stellen daher entsprechende Template-Spezialisierungen des Strukturtemplates `App` bereit. Um die verschiedenen `App`-Knotentypen nicht auch noch bei der Übersetzungsfunktion berücksichtigen zu müssen, statten wir Applikationsknoten mit einer statischen Methode `appStep` aus.

Anmerkung 10.1. Diese Optimierung ließe sich in unser Modell zurückübertragen, wenn wir zur Darstellung der Werte eines Applikationsknotens zwei neue Konstruktoren `AppF` und `AppA` einführen. Die Typdarstellung bleibt unberührt. ◊

Den Stack können wir leicht so modifizieren, daß alle Elemente an einer Wortgrenze beginnen. Dazu ersetzen wir die `sizeof`-Funktion durch ein Makro, das eine entsprechende Speicherausrichtung berücksichtigt.

Leider schreibt der C++ -Standard kein festes Speicherlayout für Strukturen vor, so daß wir im Applikationsschritt testen müssen, ob Stacklayout und Strukturlayout übereinstimmen.

Auf den Adressoperator zurückzugreifen, um die Adresse eines Datenfeldes innerhalb einer Struktur zu ermitteln, wäre jedoch ungeschickt. Der Speicherort einer Variablen ist laufzeitabhängig, so daß der Layoutvergleich zur Laufzeit erfolgen muß.

Da wir für CoreEML ausschließlich Strukturtypen mit zwei Feldern einsetzen, können wir stattdessen auf den `sizeof`-Operator zurückgreifen, der in C++ grundsätzlich zur Übersetzungszeit ausgewertet wird:

```
template <typename F,typename A>
struct Layout< App<F,A> > {
  enum { padding = sizeof(App<F,A>) - sizeof(F) - sizeof(A),
         FuncOffset = 0,
         ArgOffset = sizeof(A) + padding,
         wordAligned = ( ( (ArgOffset+1) \% sizeof(int) == 0) &&
                         (padding < sizeof(int))
                       )
       };
};
```

Die Metafunktion `Layout` bekommt einen Strukturtyp (hier einen Applikationsknoten) als Argument übergeben und errechnet das Offset der Datenfelder dieser Struktur, also deren Position, relativ zum Beginn des Speicherbereichs, in dem ein Wert mit diesem Strukturtyp liegt. Das erste Element (hier `F`) liegt standardmäßig an Offset Null. Die Lage des zweiten Elements ist abhängig vom Alignment, welches der Compiler gewählt hat. Es kann direkt

```
template <typename F,typename A>
struct App {
  // . . .
  static inline void appStep() {
    if (Layout< App<F,A> >::wordAligned)
      return;
    else {
      App<F,A> tmp = emlStack.pop<App<F,A> >();
      emlStack.push( tmp.a );
      emlStack.push( tmp.f );
    }
  };
  F f;
  A a;
};

template <int N,typename A>
struct App<Var<N>, A> {
  // . . .
  static inline void appStep() {
    // Nothing to do
  }
  A a;
};
```

Abbildung 10.6: Applikationsschritte sind abhängig vom Typ und dem Speicherlayout eines Applikationsknotens.

hinter dem ersten Element liegen, oder es kann zunächst ein Versatz (engl. *padding*) folgen, um z.B. zu garantieren, daß das zweite Datenfeld ebenfalls auf einer Wortgrenze liegt.

Für eine Struktur aus zwei Elementen läßt sich das *padding* leicht aus der Gesamtgröße der Struktur berechnen, indem man die Größe der einzelnen Elemente abzieht. Das Offset der zweiten Komponente ergibt sich dann aus der Größe der ersten und dem Versatz.

Das zweite Feld ist auf eine Maschinenwortgrenze ausgerichtet, wenn sein um eins inkrementiertes Offset durch die Größe eines Maschinenwortes teilbar ist, und das *padding* kleiner als ein Maschinenwort ist.

Mit der Metafunktion `Layout` können wir jetzt testen, ob ein Applikationsschritt tatsächlich Stackoperationen erfordert (siehe Abbildung 10.6). Die Bedingung in der `if`-Anweisung wird von den meisten Compilern zur Übersetzungszeit ausgewertet. Sollte das nicht der Fall sein, kann man mit der Metafunktion `IF` entsprechenden Programmcode zur Übersetzungszeit generieren.

10.4.2 Implementierung des Heap - *Garbage Collection*

Ein Algorithmus, der Heapzellen freigibt, die nicht mehr referenziert werden, heißt ***garbage collector***.

Referenzen auf Heapzellen können sowohl auf dem Stack, als auch in Heapspeicherzellen selbst

auftreten. Ein einfacher Algorithmus (*mark / scan* oder *mark / sweap* genannt) traversiert den Stack und markiert alle erreichbaren Speicherzellen des Heap, und solche, die von dem dort gespeicherten Term referenziert werden. In einem zweiten Schritt wird der gesamte Freispeicher durchlaufen, um alle nicht markierten Zellen freizugeben.

Ein *mark / scan* Algorithmus ließe sich in unserem Fall nur unter großem Aufwand realisieren. Einerseits können sich auch auf dem von C++ -Programmen genutzten Systemstack Referenzen auf EML-Terme befinden, andererseits läßt sich das Traversieren des Heap in der *scan*-Phase nur schwer realisieren, wenn man für Heapknoten keine einheitliche Struktur (z.B. einen Summentyp) vorschreibt.

Wir wählen daher einen anderen Ansatz, das sogenannte *reference counting* [35], [89, Seite 43 ff.]. Bei dieser Methode wird jede Heapspeicherzelle mit einem Referenzzähler ausgestattet, der inkrementiert wird, wenn eine neue Referenz auf diese Zelle hinzukommt, und dekrementiert, wenn eine Referenz verschwindet. Erreicht der Zähler den Wert Null, so wird die Speicherzelle automatisch freigegeben.

Die Frage, wie man mitbekommt, wann eine neue Referenz auf eine Heapzelle hinzukommt bzw. verloren geht, wollen wir noch ein wenig verschieben und uns zunächst überlegen, wie wir Heapzellen selbst darstellen (siehe Abbildung 10.7).

Heapzellen sind Objekte, die von der Klasse `HeapCell` abgeleitet werden. Diese stellt im Datenfeld `refCount` einen Referenzzähler bereit, der vom Konstruktor mit Null initialisiert wird. Die Methoden `attach` und `detach` regeln das Hinzukommen und Wegfallen von weiteren Referenzen auf diese Heapzelle. Geht eine Referenz verloren, so wird der Referenzzähler dekrementiert. Erreicht er den Wert Null, so wird das aktuelle Objekt mittels `delete` gelöscht.

Der Destruktor `~HeapCell` ist als virtuell vereinbart und muß von einer Kindklasse überschrieben werden (abstrakte Methode). Damit ist gewährleistet, daß beim Aufruf von `delete` in der Methode `detach` jeweils der Destruktor des aufrufenden Objekts angesprochen wird und so dessen Ressourcen freigegeben werden (Beachte: `attach` und `detach` sind *nicht* virtuell!).

```
class HeapCell {
  int refCount;
public:
  HeapCell() : refCount(0) {};
  virtual ~HeapCell() = 0;

  inline void attach() { ++refCount; }
  inline void detach() { if (- -refCount)
                           delete this;
                       }
};
```

Abbildung 10.7: Heapzellen werden als Objekte vom Typ `HeapCell` dargestellt.

Funktionsabschlüsse leiten wir von der Klasse `HeapCell` ab (siehe Abbildung 10.8). Das Klassentemplate `Closure` ist über einen Typparameter `R` parametrisierbar, der dem Ergebnistyp des Funktionsabschlusses entspricht. Ausgewertet wird der Abschluß durch Anwenden des Funktionsaufrufoperators, der hier als abstrakte Methode vorgegeben ist und daher von Kindklassen überschrieben werden muß.

```
template <typename R>
struct Closure : HeapCell {
  Closure() : HeapCell() {}
  virtual R operator()() = 0;
};
```

Abbildung 10.8: Abstrakte Klasse zur Darstellung von Funktionsabschlüssen, die sich auf dem Heap befinden.

Wir unterscheiden zwei Arten von Funktionsabschlüssen. Im einfachsten Fall handelt es sich um eine Konstante, die wir einfach in der entsprechenden Heapzelle speichern und bei Anwendung des Funktionsaufrufoperators an den Aufrufer zurückgeben:

```
template <typename T>
struct ClosureConstant : Closure<R> {
  R value;
  ClosureConstant(const R& r) : Closure<R>(), value(r) {}
  virtual R operator()() { return value; }
};
```

Echte Funktionsabschlüsse sind ein wenig aufwendiger zu realisieren, da sich ihr Zustand ändern kann (siehe Abbildung 10.9). Sie speichern entweder einen unausgewerteten CoreEML-Term, oder aber einen Wert. Bei beiden handelt es sich um Objekte, für die – wenn wir sie als Datenfeld in der Heapzelle ablegen – automatisch ein Destruktor aufgerufen würde, wenn die Heapzelle freigegeben wird. Das bedeutet aber, daß wir den unausgewerteten Term und die von ihm belegten Ressourcen erst dann freigeben könnten, wenn die zugehörige Heapzelle freigegeben wird. Besser wäre, den Term unmittelbar nach seiner Auswertung freizugeben.

Um den automatischen Destruktoraufruf zu umgehen, und explizite Kontrolle über den Lebenszyklus der Term- und Wertobjekte zu erlangen, stellen wir rohen Speicher in Form eines Byte-Feldes zur Verfügung, das groß genug ist, um eines der Objekte aufzunehmen. Den Zustand des Funktionsabschlusses (ausgewertet, oder nicht), vermerken wir im Datenfeld `evaluated`.

Der Konstruktor übernimmt einen CoreEML-Term, initialisiert das Feld `evaluated` mit `false` und kopiert den Term in den Buffer `value`. Wie beim Stack, reicht einfaches Kopieren nicht aus, da ein neues Objekt angelegt wird, muß an dieser Stelle ebenfalls auf den *placement* `new`-Operator zurückgegriffen werden.

Im Destruktor `~ClosureTerm` sind die angeforderten Ressourcen freizugeben. Je nachdem, ob der CoreEML ausgewertet wurde, rufen wir den Destruktor des Terms, oder den des berechneten Wertes explizit auf.

Wird der Wert des Funktionsabschlusses durch Anwenden des Funktionsaufrufoperators erfragt, können wir den Wert zurückgeben, sofern der Term bereits ausgewertet wurde. Ansonsten setzen wir das Flag `evaluated` auf `true` und legen den CoreEML-Term auf dem Stack ab. Jetzt können wir den Destruktor des Term-Objekts aufrufen, da dieses Objekt von nun an nicht mehr benötigt wird (eine Kopie befindet sich ja auf dem Stack) Den Ergebniswert der Berechnung erhalten wir durch Aufruf der vom Metaprogramm `translateCoreEML` generier-

```
template <typename SCRIPT,typename R,typename EXPRESSION>
struct ClosureTerm : Closure<R> {
  mutable bool evaluated;
  mutable char value[ sizeof(EXPRESSION) > sizeof(R) ?
                      sizeof(EXPRESSION) : sizeof(R) ];
  ClosureTerm(const EXPRESSION& e) : Closure<R>() {
    evaluated=false;
    new (value) EXPRESSION(e)
  }
  ~ClosureTerm() {
    if (evaluated)
      reinterpret_cast<R*>(value)->~R();
    else
      reinterpret_cast<EXPRESSION*>(value)->~EXPRESSION();
  }
  virtual R operator()() {
    if (evaluated) return *reinterpret_cast<R*>(value)
    else {
      evaluated = true;
      emlStack.push<EXPRESSION>( *reinterpret_cast<EXPRESSION*>(value) );
      reinterpret_cast<EXPRESSION*>(value)->~EXPRESSION();
      new (value) R( translateCoreEML<SCRIPT,
                                      TML::C<EXPRESSION, TML::N> >::apply()
                   );
    }
  }
};
```

Abbildung 10.9: Heapzelle zur Aufnahme von „echten" Funktionsabschlüssen.

ten Funktion `apply`. Da bei der Auswertung ein neues Objekt entsteht, initialisieren wir den Buffer wieder durch Anwenden des *placement* `new`-Operators.

Auf die Vorzüge der vorzeitigen Ressourcenfreigabe durch Aufruf des Destruktors werden wir gleich nochmal zurückkommen, wenn wir wissen, wie die eigentliche Referenzzählung funktioniert.

Zusätzliche Referenzen auf ein Objekt können in C++ durch Zuweisungen oder durch Kopierkonstruktoren entstehen. Eine Referenz wird von einem Objekt gelöst, wenn das referenzierende Objekt gelöscht wird, oder der Referenz ein anderes Objekt zugewiesen wird.

In alle Mechanismen kann prinzipiell durch Überladung des Zuweisungsoperators, Bereitstellung eines Kopierkonstruktors und Implementierung eines Destruktors eingegriffen werden; allerdings funktioniert dies nur für Objekte und nicht für Zeiger auf Objekte.

Wir packen Zeiger auf Funktionsabschlüsse daher in ein *proxy*-Objekt ein, und speichern keine Zeiger, sondern `proxy`-Objekte in CoreEML-Termen. Der Typ dieser `proxy`-Objekte entspricht einer Instanz des Klassetemplates `EMLClosure`, das als Typparameter den Rückgabetyp des Funktionsabschlusses erwartet (siehe Abbildung 10.10).

Von `EMLClosure` instanziierte Klassen stellen zwei Konstruktoren bereit. Der erste übernimmt einen Zeiger auf eine existierende Heapzelle (einen Funktionsabschluss) und ruft dessen

```
template <typename R>
struct EMLClosure {
  Closure<R>* ptr;

  EMLClosure(const Closure<R>* _ptr) : ptr(_ptr)   { ptr->attach(); }
  EMLClosure(const EMLClosure& rhs) : ptr(rhs.ptr) { ptr->attach(); }

  ~EMLClosure() { ptr->detach(); }

  EMLClosure& operator=(const EMLClosure& rhs) {
    if (ptr != rhs.ptr) {
      ptr->detach();
      ptr = rhs.ptr;
      ptr->attach();
    }
    return *this;
  }

  inline operator R() { return (*ptr)(); }
};
```

Abbildung 10.10: Abgespeckter *smart pointer* [7] zur Darstellung von Funktionsabschlüssen.

`attach`-Methode auf (es entsteht ja gerade eine neue Referenz auf diese Heapzelle). Der Kopierkonstruktor übernimmt ein bereits existierendes *proxy*-Objekt als Argument, initialisiert den Zeiger auf die Heapzelle entsprechend und ruft ebenfalls die Methode `attach` der referenzierenden Heapzelle auf.

Endet der Lebenszyklus eines Objekts vom Typ `EMLClosure<T>`, so garantiert der C++ -Standard den Aufruf des Destruktors für dieses Objekt. Wird der Destruktor aufgerufen, bedeutet dies, daß eine Referenz auf das referenzierte Objekt verloren geht und wir rufen daher dessen `detach`-Methode auf. Handelte es sich um das letzte Objekt, welches die Heapzelle `ptr` referenziert, wird dieses automatisch gelöscht (siehe Implementierung von `detach` in Abbildung 10.7).

Wird einem *proxy*-Objekt ein anderes *proxy*-Objekt zugewiesen, müssen wir den Fall der Selbstzuweisung gesondert behandeln: Da hier keine neue Referenz auf eine Heapzelle entsteht, sind keine Änderungen am Referenzzähler erforderlich.

Handelt es sich um keine Selbstzuweisung, so wird die Referenz auf die Heapzelle `*ptr` durch Aufruf der `detach`-Methode gelöst, wobei sie u.U. gelöscht wird. Anschließend wird `ptr` mit dem Heapzellenzeiger der rechten Seite initialisiert und die zugehörige `attach`-Methode aufgerufen. Um das *daisy chaining* (`a=b=c`) zu ermöglichen, wird als Ergebnis eine Referenz auf das *proxy*-Objekt selbst zurückgegeben.

Durch den Konvertierungsoperator zum Typ `R` können `EMLClosure<R>`-Objekte überall dort eingesetzt werden, wo ein Wert vom Typ `R` erwartet wird. Er wird uns später dabei helfen, C++ -Funktionen mit nicht-strikten Argumenten zu realisieren.

Betrachten wir nun nochmal die Implementierung von `ClosureTerm::operator()()`. Mit dem Ablegen des CoreEML-Terms auf dem Stack geht der Aufruf eines Kopierkonstruk-

tors einher. Sollte der Term Referenzen auf Heapzellen enthalten (z.B. ein Term vom Typ `App<App<OpPlus,EMLClosure<int> >,EMLClosure<int> >`) so werden deren Kopierkonstruktoren aufgerufen, wodurch die entsprechenden Referenzzähler erhöht werden.

Der explizite Aufruf des Destruktors des CoreEML-Term-Objekts erzwingt den Aufruf von Destruktoren für alle Datenfelder dieses Objektes. In unserem Beispiel wird daher der Destruktor der beiden `EMLClosure<int>` Objekte aufgerufen und die Referenzzähler der zugehörigen Heapzellen werden dekrementiert.

Wird im weiteren Verlauf der Berechnung ein Objekt `proxy`-Objekt vom Stack gelöscht, so kann die zugehörige Heapzelle sofort freigegeben werden.

Anmerkung 10.2. Hier werden die Referenzzähler auf ein Objekt zunächst inkrementiert (`push`) und anschließend direkt wieder dekrementiert. Als Optimierung haben wir in der realen Implementierung eine `rawPush`-Methode zur Verfügung gestellt, die kein Objekt, sondern einen Zeiger auf einen rohen Speicherbereich übernimmt. Dadurch können wir den der `push`-Methode inhärenten Kopierkonstruktor (aufgerufen durch den *placement* `new`-Operator) umgehen und können auf den expliziten Aufruf des Destruktors für den CoreEML-Term verzichten.

Dieses Vorgehen ist nur dann problematisch, wenn absolute Zeiger auf die Datenelemente eines Objekts existieren, da diese mit dem Verschieben ungültig werden – dies ist jedoch eher selten der Fall und man sollte in solchen Fällen ohnehin auf Methodenzeiger zurückgreifen, die relativ zum Beginn des Speicherbereichs eines Objekts sind (siehe [106, Seite 106 ff.]). ◊

Die Vor- und Nachteile von *garbage collection* mittels Referenzzählung wollen wir nicht unerwähnt lassen, zumal sich insbesondere im Zusammenhang mit funktionalen Sprachen Probleme ergeben können,

Referenzzählung hat den Vorteil, daß der Aufwand der *garbage collection* gleichmäßig über eine Berechnung verteilt ist – im Gegensatz zu *mark / scan* Verfahren, wo ein Programm für einen bestimmten Zeitabschnitt ausschließlich mit dem Auffinden und Löschen von nicht mehr benötigten Heapzellen beschäftigt ist. Referenzzählung eignet sich daher besonders gut für interaktive Systeme.

Nicht mehr referenzierte Objekte werden bei diesem Verfahren unmittelbar erkannt und freigegeben. Bezogen auf den Heap ist das *working set*, also das Speicherfenster, welches für eine Berechnung verwendet wird, recht klein. Im besten Fall kann der gesamte Heap über die gesamte Dauer einer Berechnung im Cache verbleiben, woraus sich ein sehr schneller Zugriff auf die Datenelemente im Heap ergibt.

Nachteilig sind der zusätzliche Aufwand zur Speicherung des Referenzzählers und die ständig notwendigen Aktualisierungen des Zählerstandes. Wie wir am Beispiel der Klasse `ClosureTerm` gesehen haben, ergeben sich mitunter unnötige Sequenzen von Inkrementierung und Dekrementierung, die man z.T. nur schwer erkennen und vermeiden kann.

Darüber hinaus kann eine Operation das Anpassen mehrerer Referenzzähler erfordern, so daß der Aufwand im Allgemeinen schwer einzuschätzen ist. Speziell unsere Implementierung bringt den Nachteil der virtuellen Destruktoren. Wird eine Heapzelle mittels `delete` gelöscht, so kann dies den Aufruf einer großen Zahl weiterer virtueller Destruktoren zur Folge haben; z.B. dann, wenn eine Heapzelle selbst Referenzen auf Heapzellen speichert. Da virtuelle Methoden indirekt über einen Methodenzeiger aufgerufen werden, ist die Wahrscheinlichkeit groß, daß es in diesem Zusammenhang zu *cache*-faults kommt. Da das Sprungziel zur Übersetzungszeit nicht bekannt ist, können virtuelle Destruktoren nicht *inline* expandiert werden, so daß Optimie-

rungspotential verloren geht – das Löschen einer Heapzelle kann demnach eine große Last für den Stack bedeuten.

Schwierig erweist sich insbesondere die Behandlung von zirkulären Datenstrukturen:

```
circularList = 1:2:3:circularList
```

Jedes Listenelement ist in einer eigenen Heapzelle abgespeichert. Am Ende der Berechnung stehen die zugehörigen Referenzzähler alle auf Eins – auch der des ersten Elements, da es vom letzten Element der Liste referenziert wird. Als Konsequenz wird die Liste niemals freigegeben.

10.4.3 Instanziieren von Superkombinatoren

Das Instanziieren eines Superkombinators ist die zeitkritischste Operation der TIM, weshalb wir versuchen, einen möglichst großen Teil bereits zur Übersetzungszeit zu erledigen (partiell auszuwerten).

Drei Metafunktionen spielen hierbei eine Rolle:

- `lazyInstantiate`: Berechnet den Typ der instanziierten rechten Seite unter Berücksichtigung der verzögerten Auswertungsstrategie.
- `getVarOffset`: Berechnet das Offset des ersten Auftretens einer instanziieren Variable im Instanziierungspuffer.
- `doInstantiate`: Erzeugt eine Funktion, die einen Superkombinator instanziiert und an einer bestimmten Adresse im Speicher (in der Regel auf dem Stack der *staged*-TIM) ablegt.

Wir wollen hier nicht die vollständige Implementierung dieser Funktionen wiedergeben, sondern uns auf die wesentlichen Teile konzentrieren.

Die Funktion `lazyInstantiate` ist mit der Funktion `instantiate`, die wir im Rahmen der Typberechnung vorgestellt haben, nahezu identisch (siehe Abbildung 9.11 auf Seite 191). Unterschiede ergeben sich lediglich bei der Instanziierung eines Superkombinators. Bei verzögerter Auswertung können Terme auf dem Stack liegen, die ausgewertet, oder aber auf den Heap verschoben werden, bevor sie substituiert werden. Abbildung 10.11 zeigt den entsprechenden Ausschnitt der Definition von `lazyInstantiate`.

Das eigentliche Erzeugen des Funktionsabschlusses wird an die Metafunktion `mkGraphNode` delegiert. Damit können wir später, durch Spezialisierung dieses Templates, auf bestimmte Ergebnis- oder Applikationsknotentypen spezialisierte Heapzellen erzeugen. Liegt eine Konstante auf dem Stack, wird eine Heapzelle vom Typ `ClosureConstant` erzeugt. Für Applikationsknoten hingegen eine Heapzelle vom Typ `ClosureExpression`. Später, bei der Integration von algebraischen Datentypen, werden wir weitere Spezialisierungen hinzufügen.

Das Instanziieren eines Superkombinators auf dem Stack ließe sich analog zum Metaprogramm `lazyInstantiate` realisieren. Die ASTs eines instanziieren Superkombinators und einer nicht-instanziieren rechten Seite werden synchron traversiert. Für jeden Knoten und jedes Blatt wird eine Funktion generiert, die den nicht-instanziierten Superkombinator und einen Zeiger auf den Speicherbereich, der mit dem instanziierten Knoten (Blatt) gefüllt werden soll, übernimmt.

```
template <typename SCRIPT,typename E,typename R>
struct mkGraphNode {
  typedef EMLClosure<R> RET;
  static inline RET apply(const E& n) {
    return EMLClosure<R>( new ClosureConstant<E>(e) );
  }
};

template <typename SCRIPT,typename F,typename A,typename R>
struct mkGraphNode<SCRIPT,App<F,A>, R> {
  typedef EMLClosure<R> RET;
  static inline RET apply(const E& n) {
    return EMLClosure<R>( new ClosureExpression<SCRIPT,E,R>(e) );
  }
};

template <typename SCRIPT,  // CoreEML-Script
          typename LHS,     // Linke Seite der Superkombinatordefinition
          int NAME,         // Name der zu substituierenden Variable
          typename STACK    // Aktueller Stack
         >
struct lazyInstantiate<SCRIPT, LHS, Var<NAME>, STACK > {
  enum { arity      = getAppDepth<LHS>::Ret,
         argpos     = getArgPos<LHS,NAME>::Pos,
         stackpos   = arity - argpos - 1,
         stackSize  = TML::Length<STACK>::Ret,
         evalArg    = getArgPos<LHS,NAME>::Strict,
         notPara    = (argpos >= arity) || (stackSize<=stackpos)
       };
  typedef typename getStackElem<STACK,stackpos>::RET StackVal;
  typedef typename translateCoreEML<SCRIPT,
                                    TML::C<StackVal,TML::N> >::RET
                                    value_t;
  typedef typename mkGraphNode<SCRIPT,StackVal, value_t>::RET Arg_t;

  typedef typename If< notPara,
                       Var<NAME>,
                       typename
                       If< evalArg,
                           evaluatedStackVal,
                           Arg_t,
                         >::RET
                     >::RET RET;

};
```

Abbildung 10.11: Berechnung des Typs eines instanziierten Superkombinators (verzögerte Auswertung).

```
template <typename N, typename F,typename A,
          typename iF,typename iA, int offset>
struct getVarOffset<N,          // Variablenname
                    App<F,A>,   // uninstanziierter Knoten
                    App<iF,iA>,// instanziierter Knoten
                    offset     // aktuelles Offset
                   > {
enum { foundInF = getVarOffset<N, F, iF, offset>::found,
       foundInA = getVarOffset<N, A, iA, offset>::found,
       found    = foundInF || foundInA,
       argOff   = Layout< App<iF,iA> >::ArgOffset + offset,
       funcOff  = Layout< App<iF,iA> >::FuncOffset + offset,
       Ret      = foundInF ? (int)getVarOffset<N,F,iF, funcOff>::Ret
                           : (int)getVarOffset<N,A,iA, argOff>::Ret
   };
};
```

Abbildung 10.12: Berechnung des Offsets des ersten Auftretens eines zu einer Variable N gehörenden Werts im Instanziierungspuffer.

Trifft man auf ein Blatt, so handelt es sich um eine C++ -Konstante, die mit Hilfe des *placement* new-Operators kopiert wird:

```
template <typename SCRIPT,  // CoreEML-Script
          typename STACK,   // Aktueller Stack
          typename SCNAME,  // Superkombinatordefinition
                            // (Sc<LHS,RHS>)
          typename VARLIST, // Liste der Parameter, die
                            // bereits besucht wurden
          typename LEAF     // Blattyp des AST
         >
struct doInstantiate {
  typedef Leaf RET;
  typedef VARLIST  nVarList;
  static inline void apply(// Wert des Blattes aus der rechten
                           // Seite der Sc-Definition
                           const LEAF& e,
                           // Zeiger auf den Puffer
                           void* where) {
    new (where) LEAF(e);
  }
};
```

Beim Instanziieren eines Superkombinators auf dem Stack müssen wir aufpassen, daß Argumente nur einmal auf den Heap verschoben werden. Taucht ein nicht-strikt auszuwertender Parameter auf der rechten Seite mehrfach auf, müssen diese im instanziierten Superkombinator auf ein- und dieselbe Heapzelle verweisen. Im Falle der strikten Argumentauswertung

```
template <typename SCRIPT,typename T,
          typename A1,typename A2>
struct translateCoreEML< SCRIPT,
                         TML::C< App<App<OpPlus.A1>, A2>, T> > {
   typedef TML::C< App<App<OpPlus,A1>,A2> node_t;
   typedef typename inferType<SCRIPT,TML::N,TML::N,A1>::RET Arg1_t;
   typedef typename inferType<SCRIPT,TML::N,TML::N,A2>::RET Arg2_t;
   typedef typename inferType<SCRIPT,TML::N,TML::N,
                              App<App<OpPlus,Arg1_t>,Arg2_t>
                             >::RET  RET;
   static inline RET apply() {
     if (Layout<node_t>::wordAligned) {
       Arg1_t firstArg  = translateCoreEML<SCRIPT,A1>::apply();
       Arg2_t secondArg = translateCoreEML<SCRIPT,A2>::apply();
       return firstArg + secondArg;
     } else {
       node_t n = emlStack.pop<node_t>();
       emlStack.push<A1>( n.getFunction().getArg() )
       Arg1_t firstArg = translateCoreEML<SCRIPT,A1>::apply();
       emlStack.push<A2>( n.getArg() );
       Arg2_t secondArg = translateCoreEML<SCRIPT,A2>::apply();
       return firstArg + secondArg;
     }
   }
};
```

Abbildung 10.13: Übersetzung der Grundoperation `OpPlus`.

dürfen Argumente nur einmal ausgewertet werden.

Wir merken uns daher in der Liste `VARLIST` die bereits behandelten Variablen. Ist eine Variable zu substituieren, die noch nicht in `VARLIST` enthalten ist, erzeugen wir je nach Auswertungsstrategie eine neue Heapzelle, oder werten die entsprechende Variable aus und schreiben den so erhaltenen Wert an die entsprechende Stelle im Instanziierungspuffer. Wird eine Variable wiederholt instanziiert, berechnen wir mit der Metafunktion `getVarOffset` das Offset ihres ersten Auftretens im Instanziierungspuffer und kopieren den bereits substituierten Wert an die neue Stelle.

Die Metafunktion `getVarOffset` kann man unter Rückgriff auf die Metafunktion `Layout` realisieren (siehe Abbildung 10.12).

10.4.4 Implementierung der Transitionsschritte

Applikationschritte haben wir bereits bei der Diskussion der Stackimplementierung vorgestellt; C++ -Konstanten werden vom EML-Stack entfernt und direkt an den Aufrufer zurückgegeben.

Abgesehen von den Booleschen Operatoren `&&` und `||` sind Grundoperationen strikt in beiden Argumenten. Stimmt das Speicherlayout des Applikationsknotens mit dem Stacklayout Standard, können die Funktionsargumente ohne weitere Manipulation am Stack ausgewertet werden. Dabei entstehen zwei temporäre Werte, die in lokalen Variablen (also auf dem

Maschinenstack) abgelegt werden.

Bei unterschiedlichem Speicherlayout wird der Applikationsknoten zunächst vom EML-Stack in eine lokale Variable verschoben, um dann die Argumente auf dem EML-Stack abzulegen und mit deren Auswertung fortzufahren (siehe Abbildung 10.13). Alternativ könnte man den Stackzeiger des EML-Stacks nach der Auswertung des ersten Arguments auch um das `padding` erhöhen, so daß er auf den Argumentteil der `App`-Struktur zeigt.

Vor der Auswertung einer Superkombinatorapplikation befinden sich die Funktionsargumente auf dem Stack. Den Superkombinator auf die Stackspitze (über die Argumente) abzulegen wäre unklug, da die Argumente bei der TIM nach der Instanziierung nicht mehr gebraucht werden (es werden ja keine Stackreferenzen in die rechte Seite eingesetzt). Es könnte ein Funktionsabschluß auf dem Stack liegen, der an einen strikt auszuwertenden Parameter gebunden wird. Bevor dieser in die rechte Seite eingesetzt wird, ist er auszuwerten; die zugehörige Heapspeicherzelle kann jedoch nicht freigegeben werden, da sie nach wie vor vom Stack referenziert wird.

Wir verfahren daher wie folgt: Die Argumente werden im Stack nach oben verschoben, so daß wir unmittelbar dahinter den instanziierten Superkombinator ablegen können. Nach dem Instanziieren können die Argumente gefahrlos vom Stack entfernt werden, womit die von ihnen belegten Ressourcen u.U. wieder verfügbar werden.

Dieses Vorgehen hat positive Auswirkungen auf den verwendeten Stackspeicherplatz und hilft, Heapknoten möglichst frühzeitig freizugeben. Für komplexe Funktionsargumente kann das Verschieben jedoch zu einer sehr aufwendigen Operation werden. Als Ausweg könnte man komplexe Funktionsargumente direkt beim Instanziieren auf den Heap verschieben, so daß keine CoreEML-Ausdrücke, sondern nur noch C++ -Konstanten und Funktionsabschlüsse auf dem Stack liegen. Der Einfachheit halber haben wir diese Optimierung in unserer prototypischen Implementierung ausgespart.

Auch bei der Auswertung von `IF` können Ressourcen vorzeitig freigegeben werden. Vor der Auswertung von `IF` liegen Bedingung, `then`- und `else`-Zweig auf dem Stack. Verzweigt der Kontrollfluß in den `then`-Zweig, können die vom `else`-Zweig gebundenen Ressourcen vor dessen Auswertung freigegeben werden. Dasselbe gilt für den `then`-Zweig, wenn der Kontrollfluß den `else`-Zweig durchläuft (siehe Abbilding 10.14).

10.4.5 Endrekursive Aufrufe

Schleifen lassen sich in EML nur über Rekursion ausdrücken. Vor jedem rekursiven Aufruf wird ein Superkombinator instanziiert und auf dem Stack abgelegt. Unmittelbar darunter befinden sich die noch unausgewerteten Teile eines zuvor instanziierten Superkombinators.

Betrachten wir die Situation am Beispiel der Fakultätsfunktion:

```
( fac(!x) = If(x == 0, 1, x*fac(x-1) ) )[10];
```

Vor der Auswertung des else-Zweiges haben die Stacks (EML und C++) die folgende Gestalt:

```
template <typename SCRIPT,typename ST,
          typename C,typename T,typename E>
struct translateCoreEML<SCRIPT,
                        TML::C< App<App<App<OpIf,C>,T>,E>, ST > {

  typedef typename inferType<SCRIPT, TML::N, TML::N,
                             App<App<App<OpIf,C>,T>,E> >::RET RET;

  static inline RET apply() {
    typedef  App<App<App<OpIf,C>,T>,E> node_t;
    // Aktuellen Stackzeiger merken
    node_t& node = emlStack::_top<node_t>::apply(EMLStack);
    // Die Bedingung liegt auf der Stackspitze
    if ( translateCoreEML<SCRIPT, TML::C<C,TML::N> >::apply()) {
      // Ressourcen des else-Zweiges freigeben
      reinterpret_cast<const E*>( &(node.getArg() ) )->~E();
      // Stackzeiger auf then-Teil bewegen
      EMLStack.sptr=(char*)&node.getFunction().getArg();
      RET r = translateCoreEML<SCRIPT,TML::C<T,TML::N> >::apply();
      // Stackzeiger hinter else-Zweig bewegen
      EMLStack.sptr=((char*)&node) + EML_STACK_ELEM_SIZE( node_t );
      return r;
    }
    else {
      // ... analog zu then-Zweig
      reinterpret_cast<const T*>( &(node.getFunction().getArg() ) )->~T();
      EMLStack.sptr=(char*)&node.getArg();
      RET r = translateCoreEML<SCRIPT,TML::C<E,TML::N> >::apply();
      EMLStack.sptr=((char*)&node) + EML_STACK_ELEM_SIZE( node_t );
      return r;
    }
  }
};
```

Abbildung 10.14: Auswertung von IF.

EML	C++
OpMult 10 fac App(App(OpMinus,10),1)	

Bei der Betrachtung des C++ -Stacks wollen wir uns im weiteren nur auf die für uns relevanten Daten konzentrieren – eventuell auf dem Stack befindliche Rücksprungadressen lassen wir außer acht.

Die anstehende Multiplikation ist strikt in beiden Argumenten. Es wird zunächst das erste Argument auf dem EML-Stack abgelegt und ausgewertet (siehe auch Abbildung 10.13). Das Ergebnis der Berechnung wird in einer lokalen Variable (also auf dem C++ -Stack) abgelegt. Anschließend wird das zweite Argument ausgerechnet. Dazu werden die obersten beiden Argumente vom Stack gelöscht und es wird mit der Auswertung fortgefahren. Die Stacksituation stellt sich dann so dar:

EML	C++
fac App(App(OpMinus,10),1)	10

Da die Fakultätsfunktion strikt in `x` ist, wird das Argument ausgewertet, bevor es in die rechte Seite eingesetzt wird. Vor der Auswertung des else-Zweiges ergibt sich folgendes Stack-Bild:

EML	C++
OpMult 9 fac App(App(OpMinus,9),1)	10

Abermals wird das erste Argument ausgewertet und auf dem C++ -Stack zwischengespeichert, bevor mit der Auswertung des zweiten fortgefahren wird:

EML	C++
fac App(App(OpMinus,9),1)	9 10

Verfolgt man den Prozeß in Gedanken weiter, so liegen irgendwann alle Zahlen von Eins bis Neun auf dem C++ -Stack – ganz zu schweigen von den jeweiligen Rücksprungadressen.

Die in CoreEML implementierte Fakultätsfunktion ist erheblich ineffizienter, als eine rekursive oder iterative C++ -Variante:

C++ iterativ	C++ rekursiv	EML	Faktor EML/ C++
1.6s	5.8s	18.7s	3.2

Abbildung 10.15: Laufzeitvergleich Fakultätsfunktion für jeweils fünf Millionen Berechnungen der Fakultät von einhundert. Übersetzt wurde mit GNU g++ (V 3.2) mit der Option -O3. Die Optimierung von endrekursiven Aufrufen wurde mit der Option -fno-optimize-sibling-calls deaktiviert. Gemessen wurde auf einem Pentium Mobile System mit 1.4 GHz und 512Mb RAM unter Windows-XP (cygwin).

```
int facRek(int x) {
  if (x==0) return 1;
  else return x*facRek(x-1);
}

int fac(int x) {
  int accu = 1;
  while (-x) accu *= x;
  return accu;
}
```

Zum einen legen wir in EML unausgewertete Ausdrücke auf dem EML-Stack ab, so daß insgesamt mehr Speicher bewegt wird, als in der rekursiven C++ -Implementierung. In C++ werden pro Rekursion zwei Maschinenworte auf dem Stack abgelegt (eines für das Argument `x` und eines für die Rücksprungadresse), wohingegen in EML jeder rekursive Aufruf das Instanziieren eines Superkombinators auf dem Stack nach sich zieht, so daß insgesamt sechs Maschinenworte auf den Stack kopiert werden (die rechte Seite enthält drei integer-Konstanten; dreimal taucht das Argument `x` auf, welches jeweils durch eine integer-Konstante ersetzt wird). Hinzu kommt, daß wir das Argument vor dem Instanziieren im EML-Stack nach oben bewegen, wodurch weitere zwei Maschinenworte im Speicher bewegt werden. Schließlich wird pro Rekursion noch das Ergebnis der Auswertung des ersten Arguments auf den C++ -Stack bewegt, so daß insgesamt neun Maschinenworte bewegt werden müssen – im Vergleich zu zweien in der rekursiven C++ -Variante.

Ein Laufzeitvergleich der rekursiv definierten Fakultätsfunktion in C++ mit der in EML fällt daher auch eher ernüchternd aus (siehe Tabelle 10.15).

Wenn wir darauf verzichten, die Argumente an einen Superkombinator im Stack nach oben zu verschieben, verringert sich die Laufzeit auf 16s – im Verlgeich zu C++ noch immer kein konkurrenzfähiges Ergebnis.

Wir haben das Optimierungspotential jedoch noch nicht voll ausgeschöpft. Überlegen wir kurz, was die iterative C++ -Variante gegenüber der rekursiven C++ -Implementierung so schnell macht (Faktor 3.6): Die iterative Variante kommt ohne Stackoperationen aus. Das Zwischenergebnis wird in der lokalen Variable `accu` akkumuliert. Die Variablen `accu` und `x` können während der gesamten Abarbeitungszeit in Maschinenregistern gehalten werden, so daß Speicheroperationen insgesamt entfallen können. Der Aufrufkontext wird hier in jeder Iteration geändert, anstatt jeweils einen neuen anzulegen und auf dem Stack abzulegen (so wie bei der rekursiven Variante).

Betrachten wir nun eine leicht modifizierte, rekursive Version der Fakultätsfunktion in EML:

```
(fac(accu)(x) = IF(x==0,accu, fac(accu*x)(x-1)) )[10];
```

Nach dem rekursiven Aufruf wird weder der Wert der Variablen `accu`, noch der der Variablen `x` benötigt.

Allgemein nennt man rekursive Aufrufe, nach deren Rückkehr der Aufrufkontext keine Rolle mehr spielt, **endrekursiv**. Die Besonderheit von endrekursiven Aufrufen kann man ausnutzen, indem man vor dem rekursiven Aufruf keinen neuen Aufrufkontext erzeugt, sondern den alten wiederverwendet und anstatt die Funktion über einen `call`-Befehl rekursiv aufzurufen (vor dessen Ausführung die Rücksprungadresse auf dem Stack abgeleget wird) mit einem direkten Sprung an den Beginn der Funktion fortfährt.

Diese Optimierung nennt man *tail call elimination* (siehe z.B. [119, Seite 461 ff.], [139, Seite 367 ff.]) und wir wollen nun beschreiben, wie man sie prototypisch in die *staged*-TIM integrieren kann. Prototypisch deshalb, weil wir im folgenden davon ausgehen, daß eine endrekursive Funktion strikt in allen Argumenten ist. Sogenannte *tail sibling calls*, bei denen ein Aufrufkontext nach dem Aufruf einer anderen Funktion nicht mehr gebraucht wird, lassen wir ebenfalls unberücksichtigt. Darüber hinaus beschränken wir uns auf rekursive Funktionen, bei denen die Abbruchbedingung über einen `if`-Term geprüft wird und lassen keine `let`-Ausdrücke in endrekursiven Funktionen zu.

Anmerkung 10.3. Terminierende rekursive Funktionen lassen sich auch ohne den Einsatz von `if` realisieren, wenn man boolesche Operatoren einsetzt:

```
isEmpty []    == true
isEmpty (h:t) = false
getTail (h:t) = t

justOneValOrEmpty l v = (isEmpty l) || (justOneValOrEmpty (getTail l) v)
```

Im Falle einer leeren Liste `l` wird das erste Argument an `||` zu `true` reduziert. Das zweite Argument trägt zum Ergebnis nichts bei und kann unausgewertet bleiben. Ist die Liste nicht leer, muß das zweite Argument reduziert werden und es kommt zum rekursiven Aufruf. ♢

Vor der Abarbeitung eines Superkombinatorschrittes, wird getestet, ob die rechte Seite das Schlüsselwort `jump` enthält. Ist dies der Fall, wird automatisch die Auswertungsstrategie gewechselt. Das Prinzip ist wie folgt:

- Alle auf dem Stack befindlichen Argumente werden vom Stack genommen, ausgewertet und auf den Stack zurückgeschrieben.
- Der Superkombinator wird nicht auf dem Stack, sondern in einem speziellen Instanziierungspuffer expandiert. Dabei werden die Variablen nicht durch Werte, sondern durch Referenzen auf die zugehörigen Stackspeicherstellen ersetzt.

  Dazu führen wir einen neuen AST-Knotentyp `StackRef<T>` ein, der einen Zeiger auf eine Stackspeicherzelle kapselt, an der ein Wert vom Typ `T` abgelegt ist. Die Typinferenzfunktionen `inferType` erweitern wir um einen neuen Behandlungsfall für Stackreferenzen und lassen sie den Typ `T` als Ergebnis zurückliefern. Mit `translateCoreEML` erzeugen wir für Stackreferenzknoten eine Funktion, die den in `StackRef<T>` enthaltenen Zeiger dereferenziert und an den Aufrufer zurückgibt.

```
typedef typename getScDefinition<SCRIPT,SCNAME>::RET ScDef;
typedef typename ScDef::Lhs_t Lhs_t;
typedef typename SCDef::Rhs_t Rhs_t;
typedef typename instantiate<SCRIPT,Lhs_t, Rhs_t, STACK>::RET instSc_t;
typedef typename inferType<SCRIPT,STACK,TML::N,instSc_t>  RET;

typedef typename mkRefStack<STACK>::RET refStack;

char instBuffer[ sizeof(instSc_t) ];
bool goOn = true;
char retBuffer[ sizeof(RET) ];

emlStack.pushList( // Generiert eine Liste von Referenzen auf
                   // die Elemente auf dem Stack
                   mkRefStack<STACK>::apply()
                 );
// Instanziieren des Superkombinators mit den gerade
// generierten Referenzen
doInstantiate<SCRIPT, refStack, SCDEF, TML::N,
              typename SCDEF::Rhs_t
             >::apply( // Das EML-Script liegt ganz
                       // unten auf dem Stack und kann
                       // über getProgram erfragt werden
                       emlStack.getProgram<SCRIPT>::
                       getScDefinition<SCNAME>().getRhs(),
                       instBuffer
                     );
// Entfernen der Referenzliste vom Stack
emlStack.popList<refStack>();

while (goOn) {
  doEvalTailCall<SCRIPT, STACK, RESULT, instSc_t>::exec(
    buffer,   goOn,  retBuffer
  );
}
```

Abbildung 10.16: Auswertung von endrekursiven Funktionen.

- In einer `while`-Schleife, die den mit `true` initialisierten Wert einer Variablen `goOn` testet, werten wir den Rumpf des Superkombinators mit einer speziellen Strategie so lange aus, bis `goOn` den Wert `false` annimmt.

Abbildung 10.16 zeigt das zugehörige Codefragment. Die Funktion `mkRefStack` erzeugt eine Polytypliste von Stackreferenzknoten, die anschließend auf den EML-Stack verschoben wird. Durch Einsatz der Funktion `doInstantiate` werden diese Stackreferenzen in die rechte Seite der Superkombinatordefinition eingesetzt, wobei der instanziierte Superkombinator im Puffer `instBuffer` erzeugt wird.

Die Funktion `doEvalTailCall` wertet die instanziierte rechte Seite unter einer geänderten Strategie aus. Sie übernimmt eine konstante Referenz auf den Instanziierungspuffer, sowie Referenzen auf die Variable `goOn` und einen Speicherbereich zur Aufnahme des Berechnungsergebnisses.

```
// C++ Konstanten werden direkt an den Aufrufer zurückgegeben
template <typename SCRIPT,typename STACK,typename EXPRESSION>
struct fastEval {
  typedef typename inferType<SCRIPT,STACK,TML::N,EXPRESSION>::RET RET;
  static inline RET apply(const EXPRESSION& e) {
  }
};

// Additionen können unter Umgehung des EML-Stacks ausgewertet
// werden
template <typename SCRIPT,typename STACK,
          typename A1,typename A2>
struct fastEval<SCRIPT,STACK,App<App<OpPlus,A1>,A2> > {
  typedef typename inferType<SCRIPT,STACK,App<OpPlus,A1>,A2>::RET RET;
  static inline RET apply(const App<App<OpPlus,A1>,A2>& app) {
    return fastEval<SCRIPT,STACK,A1>::apply( app.getFunction().getArg() ) +
           fastEval<ECRIPT,STACK,A2>::apply( app.getArg() );
  }
};

// ... weitere Spezialisierungen ...
```

Abbildung 10.17: Metaprogramm `fastEval` zur Erzeugung von Code, der CoreEML-Terme strikt auswertet.

Um EML-Stackoperationen zu vermeiden, und weil wir die Argumente eines endrekursiven Aufrufes strikt auswerten, greifen wir bei der Implementierung von `doEvalTailCall` nicht auf `translateCoreEML` zurück, sondern stellen eine spezielle Funktion `fastEval` zur Verfügung (siehe Abbildung 10.17). `fastEval` bekommt als Argument einen CoreEML-Term übergeben und wertet diesen unter Umgehung des EML-Stacks aus.

Im Normalfall (kein rekursiver Aufruf), wertet `doEvalTailCall` den als Argument übergebenen CoreEML-Term aus und weist das Ergebnis der Variablen `retVal` zu. Die Variable `goOn` wird auf `false` gesetzt, um das Ende der Rekursion zu signalisieren (siehe Abbildung 10.18 oben).

```
template <typename SCRIPT,typename STACK,
          typename RESULT, typename EXPRESSION>
struct doExecTailCall {
  static inline void apply(bool& goOn,RESULT& retVal,const EXPRESSION& e) {
    retVal = fastEvalL<SCRIPT, STACK, EXPRESSION >::apply( e );
    goOn = false;
  }
};

template <typename RESULT,typename SCRIPT,typename STACK,typename SCNAME,
          typename F,typename A>
struct doExecTailCall<RESULT,SCRIPT,STACK,SCNAME,App<F,A> > {
  enum { isTailCall = Equal< typename getOutermostApp< App<F,A> >::RET,
                             OpJump
                           >::Ret };

  struct adjust {
    static inline void apply(bool&,RESULT&,const App<F,A>& e) {
      adjustStack<SCRIPT,SCNAME,STACK, App<F,A> >::apply(
          *reinterpret_cast<STACK*>(EMLStack.sptr), e );
    }
  };

  struct evaluate {
    static inline void apply(bool& goOn,RESULT& reVal,const App<F,A>& e) {
      retVal = fastEval<SCRIPT,TML::N, App<F,A> >::apply(e);
      goOn = false;
   }
};

  static inline void apply(bool& goOn,RESULT& retVal,const App<F,A>& app) {
    If< isTailCall,
        adjust,
        evaluate
      >::RET::apply(goOn,retVal,app); }
};
```

Abbildung 10.18: Applikationsschritt bei der Auswertung endrekursiver Aufrufe.

```
template <typename SCRIPT,typename STACK,int POS,typename NODE>
truct adjustStack {};

template <typename SCRIPT,typename STACK,int POS,int NAME,typename A>
struct adjustStack<SCRIPT,STACK,POS, App<Var<NAME>, A> > {
  static inline void apply(const App<Var<NAME>,A>& restrict app) {
    getStackElem<STACK,POS>::getRef(EMLStack) =
      fastEval<SCIPT,TML::N, A>::apply(app.getArg());
  }
};

template <typename SCRIPT,typename STACK,int POS,typename F,typename A>
struct adjustStack<SCRIPT,STACK, POS, App<F,A> > {
  static inline void apply(const App<F,A>& restrict app) {
    adjustStack<SCRIPT,STACK,POS-1,F>::apply( app.getFunction() );
    getStackElem<STACK,POS>::getRef(EMLStack) =
       fastEval<SCRIPT,TML::N, A>::apply(app.getArg()); }
};
```

Abbildung 10.19: Anpassung des Stacks bei der Auswertung von endrekursiven Aufrufen.

Der Applikationsschritt bedarf einer gesonderten Behandlung. Ist das am weitesten links außen stehende Funktionsymbol `jump`, handelt es sich um einen endrekursiven Aufruf. Dann werden die Argumente an `jump` ausgewertet, und an die entsprechende Stelle im EML-Stack kopiert (Metafunktion `adjustStack` – siehe Abbildung 10.19).

Abbildung 10.20 zeigt das Ergebnis unserer Bemühungen. Das nicht-optimierte, endrekursive EML-Fakultätsprogramm ist etwas langsamer als die nicht-endrekursive Version. Dieses Ergebnis war zu erwarten, da der neue Parameter `accu` zusätzlichen Speicherplatz auf dem EML-Stack einnimmt und beim Instanziieren nun fünf Variablen zu berücksichtigen sind.

Ohne Optimierung kommt es bei der Übersetzung mit dem Glasgow Haskell Compiler zu einem Stapelüberlauf (die Testroutine, welche die Fakultätsfunktion aufruft, ist ebenfalls rekursiv, so daß die Stacklast hier sehr hoch ist).

Erlauben wir dem C++ -Übersetzer die Optimierung von endrekursiven Aufrufen, so erzeugt er Programmcode, der genauso effizient ist, wie die iterative Variante. Die optimierte, endrekursive EML-Variante ist annähernd so schnell, wie der nicht-optimierte, endrekursive C++ -Code – hier hätte man vielleicht mehr erwartet, aber vom Glasgow Haskell Compiler trennt uns nicht sehr viel.

Warum ist nun der iterative C++ -Code immer noch schneller? Ein Blick in den von g++ generierten Maschinencode offenbart, daß der Übersetzer gute Arbeit beim *inlining* geleistet hat, aber der Zugriff auf Strukturelemente über Adressberechnungen erfolgt (z.B. Funktions- und Argumentteil in der von der Metafunktion `adjustStack` generierten Funktion `apply`): Die Adresse eines Strukturobjekts wird in ein Maschinenregister geladen und ggf. entsprechend der angesprochenen Komponente inkrementiert. Der Zugriff auf die Komponente selbst verläuft indirekt und ist damit nicht sonderlich effizient. Weitere indirekte Speicherzugriffe ergeben sich im Zusammenhang mit den Stackreferenzen, die wir in den Rumpf des Superkombinators instanziiert haben.

C++ (endrekursiv)	EML (endrekursiv)	Haskell (endrekursiv)	Faktor EML/ C++ endrekursiv (nicht optimiert)	Faktor EML/Haskell
5.8s	25.5s	Stapelüberlauf	4.3	–
optimiert	optimiert	optimiert		
1.6s	5.9s	3.9s	1.01	1.5

Abbildung 10.20: Laufzeitvergleich Fakultätsfunktion für jeweils fünf Millionen Berechnungen der Fakultät von einhundert. C++ und EML wurden mit GNU g++ (V 3.2) mit der Option -O3 übersetzt. Für die optimierte C++ -Variante wurde zusätzlich die *tailcall optimization* von g++ aktiviert. Der Haskell-Testcode wurde mit dem Glagow Haskell Compiler (ghc V6.01) übersetzt. Ohne Optimierung kommt es mit ghc zu einem Stapelüberlauf. Gemessen wurde auf einem Pentium Mobile System mit 1.4 GHz und 512Mb RAM unter MS-Windows XP (cygwin).

Eine iterative Testversion, in der wir alle diese indirekten Speicherzugriffe direkt codiert haben benötigte 4.2s, wodurch wir unseren Verdacht bestätigt sehen. Offenbar fehlt es g++ an der für unsere Absichten wichtigen *lightweight object optimization*.

Ein C++ -Übersetzer, der diese Optimierung unterstützt, ist der von Kuck and Associates (kurz KAI) entwickelte Compiler KCC. Leider ist dieser nach der Aquirierung von KAI durch Intel nicht mehr verfügbar.

KCC übersetzt C++ -Code zunächst in C-Code und nutzt dann den installierten C-Compiler, um daraus Maschinencode zu generieren. Dieses Vorgehen ist für uns besonders interessant, da wir am C-Zwischencode gut ablesen können, ob unsere Optimierungen wirklich greifen.

Uns stand KCC V4.0f unter Linux V2.4.18 zur Verfügung. Als C-Compiler wurde von KCC der GNU C-compiler gcc in der Version 2.95.3 eingesetzt, der noch keine Unterstützung für die Optimierung von endrekursiven Aufrufen bietet.

Die Ergebnisse mit KCC zeigen (siehe Abbildung 10.21), daß wir mit unserem *staged interpreter* tatsächlich an das Niveau von direkt in Maschinensprache übersetztem Programmcode heranreichen. Im Vergleich zu ghc (allerdings in der Version 5.04) ist übersetzter EML-Code sogar um den Faktor 1.7 schneller.

Ein Blick in den von KCC generierten C-Zwischencode zeigt, daß *inlining* und *lightweight object optimization* zu Programmcode führen, der im Kern der iterativen C++ -Variante entspricht:

C++ (endrekursiv)	EML (endrekursiv)	Haskell (endrekursiv)	Faktor EML/ C++ endrekursiv (nicht optimiert)	Faktor EML/Haskell
16.96s	88.01s	Stapelüberlauf	5.2	–
iterativ	optimiert	optimiert	EML/ C++ iterativ	
4.6s	5.7s	10.1s	1.2	0.6

Abbildung 10.21: Laufzeitvergleich Fakultätsfunktion für jeweils fünf Millionen Berechnungen der Fakultät von einhundert. C++ und EML wurden mit KCC (V 4.0e) mit den Optionen +K3 -O3 übersetzt. Für die optimierte C++ -Variante wurde zusätzlich die *tailcall optimization* von g++ aktiviert. Der Haskell-Testcode wurde mit dem Glagow Haskell Compiler (ghc V5.04) übersetzt. Ohne Optimierung kommt es mit ghc zu einem Stapelüberlauf. Gemessen wurde auf einem Pentium III System mit 450Mhz und 512Mb RAM unter Linux (Kernel 2.2.18).

```
// ...
do
#line 4871
  if (__T142381948 != __T142382140) {
#line 4718
    __T142382012 *= __T142381948;
    __T142381948 -= __T142382076;
  } else  {
#line 4779
    (*((int *)EML::tcRes)) = __T142382012;
    (*EML::goOn) = ((char)0);
  }
while ((int)_goOn);
//...
```

Im Vergleich zu C++ kommen jedoch noch Kosten für das Erzeugen der Stackreferenzen, das Instanziieren des Superkombinators im lokalen Puffer und die Initialisierung des EML-Stacks hinzu, welche wir im obigen Codefragment ausgespart haben.

Unser Ergebnis ist insofern besonders erfreulich, als daß gcc in der Version V2.95.3 keine Optimierung von Endrekursion vornimmt. Offenbar können wir durch strukturelle Typanalyse und partielle Auswertung – im beschränkten Rahmen – nachträglich Optimierungen zu einem Übersetzer hinzufügen.

10.4.6 Aufruf von C++ -Funktionen und Methoden

C++ -Funktions- und Methodenaufrufe dürfen zwar innerhalb von EML-Termen auftreten, wie wir aber bereits erwähnt haben, werden diese vor der Abarbeitung des EML-Programmes ausgewertet. C++ -Funktionen sind in der Regel nicht mit EML-Variablen kompatibel und erwarten Tupeltypen als Argument. Um sie in EML nutzbar zu machen, muß der Programmierer eine Interface-Spezifikation bereitstellen, aus der hervorgeht, wie viele Argumente die Funktion / Methode erwartet, in welchem Argument sie strikt bzw. nicht-strikt ist; welchen

```
struct EMLI_atan2 {
  enum { arity = 2; }

  template <int N>
  struct isStrict { enum { Ret = true }; }

  template <typename T1,typename T2>
  struct inferType;
  template <>
  struct inferType<float,float> {
    typedef float RET;
  };
  struct inferType<double,double> {
    typedef double RET;
  }

  static inline double exec(double a.double g) {
    return atan2(a,b);
  }
  static inline float exec(float a,float b) {
    return atan2f(a,b);
  }
};
```

Abbildung 10.22: Interfacespezifikation zur Verwendung der Funktionen `atan2` und `atan2f` aus der `C`-Mathematikbibliothek.

Ergebnistyp sie für bestimmte Argumenttypen generiert und wie sie auszuwerten ist.

Abbildung 10.22 zeigt eine entsprechende Spezifikation für die `atan2`-Funktion aus der Mathematikbibliothek. Die Stelligkeit der Funktion wird über einen Aufzählungstyp exportiert. Die Striktheitsinformation wird über die Metafunktion `isStrict` verfügbar gemacht, die Typberechnung über die Metafunktion `inferType`. Die statische Methode `exec` legt schließlich das Laufzeitverhalten fest.

Mit der Spezifikation `EMLI_atan2` kann die `atan2`-Funktion jetzt wie folgt aus EML heraus aufgerufen werden:

```
( m(x)(y) = callI<EMLI_atan2>::apply(x)(y) )[ m(2.0)(3.0) ]
```

Dabei ist `callI` ein Strukturtemplate, welches eine Interfacespezifikation als Typargument übernimmt und ein statisches Methodentemplate `apply` exportiert, welches aus dem Argument einen Applikationsknoten generiert. Wird `apply` wie in unserem Beispiel auf ein integer-Argument angewendet, so entsteht ein AST-Knoten vom Typ

```
App<App<OpCall,EMLI_atant2>,int>
```

Interfacespezifikationen für monomorphe C++ -Funktionen zu erstellen ist aufwendig. Außerdem ist klar, daß eine C++ -Funktion strikt in allen Argumenten ist. Darüber hinaus läßt

sich der Typ einer Funktion unter Einsatz von Templates leicht aus dem zugehörigen Funktionszeiger extrahieren. Um dem Programmierer das Erstellen von Schnittstellenspezifikationen zu ersparen, stellt EML daher eine Reihe von überladenen `call`-Funktionen bereit, die einen Funktionszeiger als Argument übernehmen, daraus die relevanten Typinformationen extrahieren und durch Template-Instanziierung automatisch eine passende Spezifikation erzeugen. Der Aufruf der Funktion `atan2` vereinfacht sich damit zu

```
( m(x)(y) = call(atan2)(x)(y) )[ m(2.0)(3.0) ]
```

Die Typinferenzfunktion von CoreEML läßt sich ohne größeren Aufwand um das Kommando `OpCall` erweitern: Trifft man bei der Typberechnung auf einen Knotentyp `App<OpCall,ISPEC>`, so wird mit Hilfe der Interfacespezifikation `ISPEC` geprüft, ob genügend Argumente auf dem Stack liegen. Wenn ja, so wird die Berechnung des Ergebnistyps an die in `ISPEC` definierte Metafunktion `inferType` delegiert. Dabei müssen wir berücksichtigen, daß auf dem Stack unausgewertete Terme liegen können, deren Ergebnistyp vor dem Delegieren mit der globalen `inferType` Metafunktion zu berechnen ist. Steht keine ausreichende Zahl an Argumenten bereit, wird ein Knoten vom Typ `PaI` erzeugt, der – ähnlich wie Knoten vom Typ `PaSC` – die bereits vorhandenen Argumente und die Spezifikation `ISPEC` speichert. Abbildung 10.23 zeigt einen Ausschnitt der entsprechenden C++ -Metafunktionen.

Bei der Erweiterung von `translateCoreEML` muß die in der Interfacespezifikation enthaltene Striktheitsinformation berücksichtigt werden; d.h. bevor die in der Spezifikation vorgegebene statische Methode `exec` ausgeführt wird, sind die auf dem Stack befindlichen Argumente entweder auszuwerten, oder in Funktionsabschlüsse umzuwandeln.

Wir verzichten auf die Darstellung der Erweiterung von `translateCoreEML`, da sie sich mit dem Metaprogramm `callExec` (siehe Abbildung 10.24) analog zur Erweiterung von `inferType` durchführen läßt.

Einige der gerade vorgestellten Metaprogramme sind abhängig von der Stelligkeit der aufzurufenden Funktion / Methode. Um ein hohes Maß an Flexibilität gewährleisten zu können, bietet die EML-Distribution Codegeneratoren an, die ausgehend von der maximal zu unterstützenden Stelligkeit von Funktionen, entsprechenden Programmcode generieren.

Zwei Fragen bleiben zu beantworten:

1. Wie kann man in C++ von verzögerter Auswertung profitieren; bzw. wie sieht eine C++ -Funktion / Methode aus, die nicht-strikt in einem Argument ist?
2. Wie ruft man eine Methode aus EML heraus auf?

Die erste Frage läßt sich sehr leicht beantworten: Entweder übernimmt die C++ -Funktion direkt einen Wert vom Typ `EMLClosure<T>`, oder man realisiert sie als Funktionstemplate, so daß sie mit diesem Typ kompatibel ist. Aus Benutzersicht verhält sich ein `EMLClosure<R>`-Objekt Dank des Konvertierungsoperators nach `R` ja genauso wie ein Wert vom Typ `R`:

```
double lazyCppFunc(bool condition,
                   const EMLClosure<double>& arg1,
                   const EMLClosure<double>& args
                  ) {
   if (condition) return arg1 / 2; else return arg2 / 3;
}
```

```
template <int N,typename SCRIPT.typename STACK,typename SPEC>
struct callInferType;

template <typename SCRIPT,typename STACK,typename ISPEC>
struct<0,SCRIPT,STACK,ISPEC> callInferType {
  typedef typename ISPEC::inferType::RET RET;
};

template <typename SCRIPT,typename STACK,typename ISPEC>
struct<1,SCRIPT,STACK,ISPEC> callInferType {
// A1 ist das nicht ausgewertete Argument auf dem Stack
 // (u.U. also ein CoreEML-Term)
 typedef typename STACK::Head_t A1;
// RA1 ist der Typ des ausgewerteten Arguments.
 typedef typename inferType<SCRIPT,TML::N,TML::N,A1>::RET RA1;
typedef typename ISPEC::template inferType<RA1>::RET;
};

template <typename SCRIPT,typename STACK,typename ISPEC>
struct<2,STACK,ISPEC> callInferType {
  typedef typename STACK::Head_t         A1;
  typedef typename STACK::Tail_t::Head_t A2;
  typedef typename inferType<SCRIPT,TML::N,TML::N,A1>::RET RA1;
  typedef typename inferType<SCRIPT,TML::N,TML::N,A2>::RET RA2;
  typedef typename ISPEC::template inferType<RA1,RA2>::RET;
};
// usw.

template <typename ISpec>
struct inferType<SCRIPT,STACK,CALLTRACE,App<OpCall,ISPEC> > {
  enum { arity = ISPEC::arity,
         stacksize = TML::Length<STACK>::RET
       };
  typedef typename
  If< stacksize >= arity,
      typename callInferType<arity,SCRIPT,STACK,ISPEC>::RET,
      PaI<STACK,ISPEC>
    >::RET RET;
};
```

Abbildung 10.23: Berechnung des Ergebnistyps einer in EML importierten Funktion.

```
template <typename SCRIPT,
          typename STACK,
          typename E,  // Argument auf dem Stack
          typename R,  // Typ des ausgewerteten Arguments
          bool strict  // Striktheitsflag
         >
struct convertArg { // nicht-strikte Auswertung
  typedef EMLClosure<R> RET;
  static inline RET apply() {
    return EMLClosure<R>(
      new ClosureTerm<SCRIPT,R,E>( emlStack.pop<E>() ) );
  }
};

template <typename SCRIPT,
          typename STACK,
          typename E,
          typename R
         >
struct convertArg<SCRIPT,STACK,E,R,true> {// strikte Auswertung
  typedef R RET;
  static inline RET apply() {
    return translateCoreEML<SCRIPT,TML::C<E,TML::N> >:;apply();
  }
};

template <int N,typename SCRIPT,typename STACK,typename ISPEC>
struct callExec;

template <typename SCRIPT,typename STACK,typename ISPEC>
struct callExec<0,SCRIPT,STACK,ISPEC> {
  typedef typename callInferType<0,SCRIPT,STACK,ISPEC>::RET RET;
  static inline RET apply() { return ISPEC::exec(); }
};

template <typename PROGRAM,typename STACK,typename ISPEC>
struct callExec<1,STACK,ISPEC> {
  typedef typename callInferType<1,SCRIPT,STACK,ISPEC>::RET RET;
  typedef typename callInferType<1,SCRIPT,STACK,ISPEC>::A1  A1;
  typedef typename callInferType<1,SCRIPT,STACK,ISPEC>::RA1 RA1;
  static inline RET apply() {
    return ISPEC::exec(
      convertArg<SCRIPT,STACK,
                 A1,RA1,
                 ISPEC::template isStrict<1>::Ret>::apply();
    );
  }
};

// usw.
```

Abbildung 10.24: Aufruf einer C++ -Funktion.

Der Konvertierungsoperator, der die Auswertung des Funktionsabschlusses bewirkt, wird für `arg1` nur dann aufgerufen, wenn die Bedingung `condition` wahr ist. `arg2` bleibt unausgewertet.

Die zweite Frage läßt sich ebenso leicht beantworten, wenn man sich klar macht, daß jede n-stellige Methode im Grunde genommen eine $n+1$-stellige Funktion ist. Implizit bekommt eine Methode im Parameter `this` ja einen Zeiger auf das aufrufende Objekt übergeben. Diese Idee macht man sich zu Nutze, indem man den impliziten `this`-Zeiger in der Interfacespezifikation zu einem expliziten Parameter macht:

```
class A {
  int value;
public:
   // Konstruktoren etc.
   int method(int j) const { return v + j; }
};

struct EMLI_A__getVal {
  enum { arity = 2; }
  template <typename T1,typename T2>
  struct inferType { typedef int RET; }
  static inline RET apply(const A& a,const int& j) {
    return a.method(j);
  }
};
```

Kapitel 11

Algebraische Datentypen für EML

Im Sinne eines Datenaustausches zwischen eingebetteter Sprache (EML) und Hostsprache (C++) müssen die Instanzen eines algebraischen Typs in beiden Sprachen erzeugbar und manipulierbar sein. In diesem Kapitel wollen wir schrittweise herleiten, wie man zu einer solchen Darstellung kommt.

Dabei müssen wir berücksichtigen, daß wir in EML Typberechnungen vornehmen müssen. Es reicht daher nicht, algebraische Datentypen in das Typsystem von C++ abzubilden. Zusätzlich müssen wir eigene Routinen zur Typüberprüfung von EML bereitstellen. Auf die Integration in EML werden wir erst später eingehen. Zunächst wollen wir überlegen, wie man algebraische Datentypen überhaupt in C++ darstellen könnte.

11.1 Darstellung von algebraischen Datentypen als Klassenhierarchie

Machen wir uns zu Beginn nochmal kurz den Aufbau eines algebraischen Datentyps am Beispiel von Haskell klar:

```
data List a = C a (List a) | N
```

`List` ist ein Typkonstruktor, mit dem wir neue Typen erzeugen können. `C` und `N` sind Wertkonstruktoren, mit denen ein Wert vom Typ `List a` generiert werden kann.

Mit welchem Konstruktor ein Wert vom Typ `List a` tatsächlich erzeugt worden ist, kann man aus dem Typ nicht ablesen, so daß eine entsprechende Analyse (*case analysis*) erst zur Laufzeit erfolgen kann.

Drei Operationen sind im Zusammenhang mit algebraischen Datentypen essentiell:

- `pack<C>`: Erzeugt durch Anwendung des Konstruktors `C` einen neuen Wert.
- `is<C>`: Prüft, ob der Wert eines algebraischen Typs mit dem Konstruktor `C` erzeugt wurde.
- `unpack<C,N>`: Übernimmt einen Wert mit algebraischem Typ, zerlegt ihn in seine Komponenten und gibt den Wert der `N`-ten Komponente zurück (Projektion auf die `N`-te Komponente).

`pack` entspricht in Haskell dem direkten Aufruf eines Konstruktors. Hinter dem *pattern matching* bzw. der *case analysis* verbergen sich die Funktionen `is` und `unpack`.

Monomorphe algebraische Typen können in C++ recht einfach auf eine Klassenhierarchie abgebildet werden – eine Technik, die bei der Übersetzung von funktionalen Sprachen in die Sprache einer objektorientierten virtuellen Maschine gern angewendet wird; z.B. für Mondrian [133], Generic JAVA [19], oder PIZZA [128]. In diesen Sprachen wird die Typüberprüfung vom Compiler übernommen, so daß der erzeugte objektorientierte Code als typsicher eingestuft werden kann (zumindest, was die algebraischen Typen anbelangt).

Unsere Situation ist ein wenig komplexer, da wir ohne eigenen Typchecker auskommen müssen. Betrachten wir aber zunächst ein Beispiel, das zeigt, wie man einen einfachen algebraischen integer-Listentyp mit Klassen und Objekten repräsentieren kann (siehe Abbildung 11.1).

Haskell

```
data List a = C a (List a) | N
```

C++

```
#include <typeinfo>

template <typename T>       struct pack;
template <typename T,int N> struct unpack;
template <typename T>       struct is;

struct IntList {
  IntList(IntList* v) : value(v) {}
  IntList(const IntList& rhs) : value(rhs.value) {}
  IntList* value;
protected:
  IntList() : value(0) {}
};

class C : public IntList {
  friend struct pack<C>;
  friend struct unpack<C,0>;
  friend struct unpack<C,1>;
private:
  C(int h,const IntList& t) : head(h),tail(t) {}
  int      head;
  IntList  tail;
};

class N : public IntList {
  friend struct pack<N>;
  friend struct unpack<N,0>;
private:
  N() {}
};
```

Abbildung 11.1: Abbildung des monomorphen algebraischen Datentyps `intList` auf eine Klassenhierharchie.

Typkonstruktor und Wertkonstruktoren sind zu Klassen geworden, wobei die **Wertkonstruktorklassen** von der **Typkonstruktorklasse** abgeleitet wurden. In der Typkonstruktorklasse

ist ein Zeiger auf einen Wert vom Typ `IntList` hinterlegt, der auf Grund der Inklusionsbeziehung der Klassen `N`, `C` und `IntList` sowohl auf Objekte vom Typ `N`, als auch auf Objekte vom Typ `C` zeigen kann.

Wie bei algebraischen Typen auch, müssen wir den eigentlichen Typ eines durch Konstruktorapplikation entstandenen Wertes verbergen bzw. so anpassen, daß ein Wert des zugehörigen Summentyps entsteht. Die Konstruktoren der Klassen `N` und `C` wurden daher als `privat` vereinbart.

Der einzige Weg ein Objekt vom Typ `IntList` zu erstellen, führt über den Aufruf einer mit der Metafunktion `pack` generierten Funktion. Da die Klasse `pack` für beide Konstruktorklassen als `friend` vereinbart wurde, hat sie vollen Zugriff auf die privaten Elemente der Klasse und kann folglich auch deren Konstruktoren aufrufen. Für die Klasse `pack` stellen wir folgende Template-Spezialisierungen bereit:

```
template <>
struct pack<C> {
  static inline IntList apply(int h,const IntList& t) {
    return IntList( new C(h,t) );
  }
};

template <>
struct pack<N> {
  static inline IntList apply() {
    return IntList( new N() );
  }
};
```

Der eigentliche Wert wird mit `new` in den Freispeicher verschoben, um anschließend ein `IntList`-Objekt mit dem dabei gewonnenen Zeiger zu initialisieren[1].

Der Effekt des Aufrufs `pack<C>::apply(1, pack<N>::apply() )` entspricht exakt der Konstruktorapplikation `C 1 N` in Haskell: Es entsteht ein Wert, vom Summentyp `IntList`.

Zur Umsetzung der Funktion `is<C>` greifen wir auf das Laufzeittypinformationssystem von C++ zurück. Durch Anwendung des eingebauten Operators `typeid` auf einen Typ oder einen Ausdruck erhält man einen eindeutigen Typdeskriptor. Typdeskriptoren können miteinander verglichen werden, so daß man sehr leicht prüfen kann, ob ein Objekt wirklich einen bestimmten Typ hat:

[1] Die angeforderten Systemressourcen werden in unserem Modell nicht freigegeben – diese Lücke werden wir erst später schließen.

```
template <>
struct is<C> {
  static inline bool apply(const IntList& l) {
    return typeid(l.value) == typeid(C);
  }
};

template <>
struct is<N> {
  static inline bool apply(const IntList& l) {
    return typeid(l.value) == typeid(N);
  }
};
```

Anmerkung 11.1. Um Verwirrung vorzubeugen: `typeid` liefert immer den Laufzeittyp eines Objekts und nicht den für den Compiler sichtbaren, statischen Typ. ◊

Um auf den in `IntList` gekapselten Wert zuzugreifen, bedarf es einer Typumwandlung, die entgegen der Klassenhierarchie läuft. Hat man z.B. durch Anwendung von `is<C>` herausgefunden, daß sich hinter einem Listenwert ein mit dem Konstruktor `C` erzeugter Wert verbirgt, muß man den Zeiger `value` so behandeln, als ob er auf ein Objekt vom Typ `C` zeigen würde. Man spricht von einem ***downcast***, da ein Objekt, dessen Typ in der Klassenhierarchie höher steht, in ein Objekt umgewandelt werden soll, dessen Typ weiter unten in der Klassenhierarchie angesiedelt ist.

Für *downcasts* stellt C++ den Operator `dynamic_cast<T>` bereit. Dieser prüft, ob der Laufzeittyp eines Objekts mit dem Zieltyp `T` verträglich ist. Ist dies nicht der Fall, wird ein Ausnahmefehler ausgelöst[2].

Die Funktion `unpack` läßt sich unter Anwendung von `dynamic_cast` leicht realisieren, und erlaubt uns, auf einen Test, der sicherstellt, daß wirklich auf einen passenden Konstruktorwert zugegriffen wird, zu verzichten:

[2]Tatsächlich wird ein Ausnahmefehler nur dann ausgelöst, wenn T ein Referenztyp (z.B. int&) ist. Für Zeigertypen wird im Fehlerfall der Wert 0 zurückgeliefert.

```
template <>
struct unpack<C,0> {
  static inline int apply(const IntList& l) {
    return (*dynamic_cast<C*>(l.value)).head;
  }
};

template <>
struct unpack<C,1> {
  static inline IntList& apply(const IntList& l) {
    return (*dynamic_cast<C*>(l.value)).tail;
  }
};

template <>
struct unpack<N,0> {
  static inline N& apply(const IntList& l) {
    return (*dynamic_cast<N*>(l.value));
  }
};
```

Für andere monomorphe Datentypen lassen sich in ähnlicher Weise Klassen und Template-Spezialisierungen finden.

Leider können wir den komfortablen *pattern matching*-Mechanismus nicht direkt auf C++ abbilden, so daß die Arbeit mit algebraischen Datentypen ein wenig an Attraktivität einbüßt. Die Gegenüberstellung der Implementierung der Funktion `length` in Haskell und C++ (siehe Abbildung 11.2) macht dies deutlich.

Haskell

```
length n = case n of
             N     -> 0
             C h t ->  1 + length t
```

C++

```
int length(const intList& l) {
  if (is<N>::apply(l)) return 0;
  else return 1 + length( unpack<C,1>::apply(l) );
}
```

Abbildung 11.2: Implementierung der Funktion `length` in Haskell (oben) und C++ (unten).

11.2 Probleme mit polymorphen algebraischen Datentypen

Geht man über zu polymorphen Listen, stößt man an die Grenzen der Abbildung von algebraischen Datentypen auf Klassenhierarchien.

Aus der Klasse `IntList` müßte ein Template mit einem Typparameter entstehen, in den später der Elementtyp der Liste eingesetzt wird. Zwangsläufig müssen dann aber auch die

abgeleiteten Konstruktorklassen zu Templates werden, da sie die Superklasse mit geeigneten Typargumenten instanziieren müssen.

Das Problem ist, daß im monomorphen Fall die Korrespondenz von

$$\texttt{IntList} = \texttt{C} + \texttt{N}$$

mit

$$\texttt{C} \subseteq \texttt{IntList}\ ;\ \texttt{N} \subseteq \texttt{IntList}$$

bijektiv ist, für polymorphe Listen jedoch nicht. Der Grund liegt darin, daß der Konstruktor `N` von keinem Typparameter abhängt, die Wertkonstruktorklasse `N<T>` aber sehr wohl.

Anders ausgedrückt gibt es im polymorphen algebraischen Datentyp genau ein Objekt `N` vom polymorphen Typ `IntList a`. In der Darstellung als Objekt einer Wertkonstruktorklasse hat jede Instanz von `N` jedoch einen anderen monomorphen Typ – C++ kennt keine real existierenden polymorphen Objekte.

Der Verlust dieser Korrespondenz hat schwerwiegende Auswirkungen. Zum Beispiel läßt sich eine Funktion, die eine leere Liste (also den Wert `N` erzeugt), nicht mehr implementieren, da zur Instanziierung der Konstruktorklasse `N` der Elementtyp benötigt wird – `N` ist hier nicht polymorph!

Will man mit *einem* Symbol `N` auskommen, müßte die zugehörige Konstruktorklasse von allen nur denkbaren Listentypkonstruktorklassen abgeleitet werden – das kann man in C++ jedoch nicht ausdrücken.

Alternativ könnte man auf die vom C++ -Typsystem garantierte Typsicherheit verzichten und mit `void` Zeigern arbeiten. Diese Strategie wird bei der Übersetzung von funktionalen Sprachen auf virtuelle objektorientierte Maschinen verfolgt, allerdings verwendet man hier nicht `void`, sondern das Größte aller Objekte (`Object` in JAVA) und man verliert nicht an Typsicherheit, da der Übersetzer den Ausgangscode ja bereits einer Typüberprüfung unterzogen hat.

Dieser Ansatz kommt für uns daher nicht in Betracht. Spätestens bei der Integration von algebraischen Datentypen in EML, sind wir auf die Typinformation angewiesen, um den Ergebnistyp einer Operation berechnen zu können.

Bevor wir unsere Lösung vorstellen, wollen wir uns an einem Beispiel nochmal die Eigenschaften von polymorphen algebraischen Datentypen vor Augen halten – insbesondere, was Typen anbelangt.

Gegeben sei der folgende Typ `FOO`:

```
data FOO a b = A a | B b | C b a
```

Die Haskell Funktion

```
genA a = A a
```

ist vom Typ `a -> FOO a b`; d.h., wenn wir sie mit einem Wert vom Typ `Int` aufrufen, erhalten wir einen Wert vom Typ `FOO Int b`. Dieser Wert kann problemlos zu einer Liste vom Typ `List (FOO Int Int)`, oder einer Liste vom Typ `List (FOO Int Float)` hinzugefügt werden

– ein mit dem Konstruktor `A` generierter Wert ist ja polymorph, da nicht alle Typvariablen von `FOO` durch konkrete Typargumente ersetzt wurden.

Nachfolgend nennen wir die Instanz eines polymorphen algebraischen Datentyps, die noch Typvariablen enthält, eine **polymorphe Instanz**.

Versuchen wir, diesen Zusammenhang ein wenig formaler zu beschreiben: Sei ADT der Name eines algebraischen Datentyps mit n Typparametern, T eine Menge von konkreten Typen und α eine Menge von Typvariablen. Eine polymorphe Instanz von ADT ist eine partielle Abbildung $I : \mathbb{N} \to T \cup \alpha$, die jedem Typparameter eine Typvariable oder einen Typ zuweist.

Eine polymorphe Instanz I_1 kann als Stellvertreter einer polymorphen Instanz I_2 auftreten (in Zeichen $I_1 <:_{ADT} I_2$) wenn gilt:

$$\begin{aligned} I_1 <:_{ADT} I_2 := \forall i \in \{1, \cdots, n\} : \quad & I_1(i) = I_2(i) \\ \vee \quad & \{I_1(i), I_2(i)\} \subseteq \alpha \\ \vee \quad & (I_1(i) \in \alpha) \wedge (I_2(i) \in T) \end{aligned}$$

Ähnlich wie die Untertyprelation $<:$ (siehe Seite 177, ist $<:_{ADT}$ durch eine Inklusionsbeziehung der durch die Typen repräsentierten Mengen motiviert. Für $<:$ gilt

$$\forall t \in T : \left\{t' | t <: t'\right\} \text{ ist endlich,}$$

da eine Klasse nur endlich viele Superklassen haben kann. Für $<:_{ADT}$ kann diese Eigenschaft wegen der Existenz von polymorphen Instanzen nicht gelten – ein weiteres Argument, warum wir hier mit Vererbung nicht weiter kommen.

11.3 Kodierung der Typdeklaration

Typ- und Wertkonstruktoren werden wir auf Klassentemplates abbilden, die in keinerlei Vererbungsbeziehung stehen. Damit geht uns die unmittelbare „Verdrahtung" von Wert- und Typkonstruktor verloren, die uns durch Ausnutzung von Vererbung gewissermaßen geschenkt wurde.

Da wir die Zugehörigkeit eines Konstruktors zu einem Typ spätestens bei der Typüberprüfung brauchen, codieren wir sie im C++ -Typsystem. Die Idee ist recht einfach: Einen algebraischen Datentyp beschreiben wir durch einen Typdeskriptor, der eine eindeutige ID und die auf der linken Seite der Typdefinition vereinbarten Typparameter speichert. In Konstruktordefinitionen hinterlegen wir eine Referenz auf den Typdeskriptor und versehen ihn mit einer ID, die innerhalb dieses Typs eindeutig ist.

Die Verbindung über eine ID reicht jedoch noch nicht aus. Um anhand der ID alle Konstruktoren bzw. den Typdeskriptor eines Typs zu ermitteln, fassen wir Typdeskriptor und eine Liste aller Konstruktordefinitionen in einer speziellen Klasse zusammen, die wir `DefineSumType` nennen.

Schauen wir uns dazu an einem Beispiel an, wie man einen polymorphen Typ `IntList` definieren kann. Typvariablen stellen wir genau wie funktionale Variablen dar, zur besseren Unterscheidung benutzen wir jedoch den Strukturnamen `TVar` (siehe Abbildung 11.3).

Typkonstruktoren definiert man als Instanzen des Klassentemplates `TypeCons`, das als erstes Argument eine programmweit eindeutige ID und anschließend eine (nahezu) beliebig lange Liste von Typvariablen erwartet.

```
typedef TVar<1> TA;

typedef TypeCons<1, TA>                        listDesc;
typedef ValueCons0<listDesc,0>                 N;
typedef ValueCons2<listDesc,1, int, listDesc> C;

template <>
struct DefineSumType< 1 > {
  typedef listDesc TypeDesc;
  typedef Sum<N,C> SumType;
};
```

Abbildung 11.3: Definition eines polymorphen Listentyps.

Typkonstruktoren dienen uns ausschließlich zum Transport von Typinformation, so daß das Klassentemplate `TypeCons` keinerlei Programmcode enthält:

```
struct NOTYPE {};

template <int N, A1=NOTYPE, A2=NOTYPE, A3=NOTYPE>
struct TypeCons {
  enum { typeId = N };
  typedef A1 P1;
  typedef A2 P2;
  typedef A3 P3;
};
```

Mit diesem Klassentemplate können wir algebraische Datentypen beschreiben, die von bis zu drei Typparametern abhängig sind. Lassen wir einen Typparameter bei der Template-Instanziierung weg, so ergänzt der C++ -Übersetzer automatisch den Type `NOTYPE`. So können wir durch Einsatz der Metafunktion `Equiv` später leicht testen, wie viele Typvariablen tatsächlich angegeben wurden.

Wertkonstruktorklassen werden von der Klasse `ConsCell` abgeleitet. `ConsCell` ist eine Erweiterung der Klasse `HeapCell` um ein Feld `consId` zur Aufnahme der ID des Wertkonstruktors und eine virtuelle Methode `getConsRef`, die einen Zeiger auf den Konstruktorwert zurückliefert:

```
class ConsCell : public HeapCell {
  int consId;
public:
  ConsCell(int id=-2) : consId(-2),HeapCell() {}
  ~ConsCell {}
  inline int getId() const { return id; }
  virtual ConsCell* getConsRef() const { return this; }
};
```

Die Definition von `getConsRef` scheint an dieser Stelle vielleicht überflüssig, da man die Adres-

se eines Objekts auch unmittelbar ermitteln kann. Wir müssen jedoch berücksichtigen, daß die Komponenten eines Konstruktors verzögert ausgewertet werden und sich hinter einer Konstruktorzelle eine Berechnung (ein Funktionsabschluss) verbergen kann, der ausgewertet werden muß, bevor man eine Referenz auf die Konstruktorzelle zurückgeben kann – später, bei der Besprechung der Integration von algebraischen Datentypen in EML, werden wir darauf nochmal zurückkommen.

Die vollständige Definition der Wertkonstruktorklassen können wir an dieser Stelle noch nicht angeben, da bisher nicht geklärt ist, wie wir die Werte eines algebraischen Typs darstellen. Im Prinzip stellt eine Konstruktorklasse für jede Konstruktorkomponente ein Datenfeld bereit und die Konstruktoren (im objektorientierten Sinne) sind als privat vereinbart, um zu verhindern, daß man Werte von diesem Typ direkt erzeugen kann. Prototypisch sieht die Implementierung eines zweistelligen Konstruktors so aus:

```
template <typename DESC,typename N,typename A0,typename A1>
class ValueCon2 : ConsCell {
  enum { consId = N,
         typeId = DESC::typeId
       };
  typedef DESC TypeDesc;
  friend struct pack<ValueCons2>;
  friend struct unpack<ValueCons2,0>;
  friend struct unpack<ValueCons2,1>;

  typedef typename mkConsArg<A0>::RET C0;
  typedef typename mkConsArg<A1>::RET C1;

private:
  ValueCons2(const C0& _c0,const C1& _c1) : c0(_c0),c1(_c1) {}

  C0 c0;
  C1 c1;
};
```

Der einzige Weg zur Erzeugung eines Konstruktors führt über einen Aufruf der von der Metafunktion `pack` erzeugten Funktion `apply`. Die Werte der Konstruktorkomponenten lassen sich ebenfalls nicht direkt, sondern über die von der Funktion `unpack` generierte Funktion `apply` ermitteln.

Die Metafunktion `mkConsArg<T>` berechnet in `RET`, wie der Typ `T` innerhalb der Konstruktorzelle zu speichern ist und generiert in `apply` eine Funktion, die aus einem Wert vom Typ `T` einen Wert vom Typ `mkConsArg<T>` erzeugt.

Da es in C++ keine polymorphen Werte gibt, muß ein polymorpher algebraischer Datentyp instanziiert werden, bevor er verwendet werden kann. Dazu steht die Template-Metafunktion `TypeInstantiate` bereit, die einen Typdeskriptor und eine Liste von konkreten Argumenttypen übernimmt und in `RET` einen Typdeskriptor generiert, in dem die Typvariablen entsprechend ersetzt wurden.

Zum Beispiel liefert

```
TypeInstantiate<listDesc,int>::RET;
```

den Typdeskriptor `TypeCons<1,int>` .

Da man später, beim Aufruf von `pack` und `unpack` monomorphe Konstruktoren benötigt, ist `TypeInstantiate` auch auf Wertkonstruktordefinitionen anwendbar.

11.4 Darstellung der Werte eines algebraischen Datentyps

Die Werte eines algebraischen Datentyps stellen wir als Instanz eines aus dem Klassentemplate `ADTValue<T>` gewonnenen Typs dar, wobei der Typparameter `T` durch den Typdeskriptor des algebraischen Typs ersetzt wird.

`ADTValue` ist der Klasse `EMLClosure`, die wir zur Darstellung von Funktionsabschlüssen eingesetzt haben (siehe Seite 213), in vielen Punkten ähnlich. Sie kapselt einen Zeiger auf eine Konstruktorzelle und übernimmt das Management der Referenzzählung.

Ein `ADTValue<T>`-Objekt kann nicht nur mit Objekten des gleichen Typs initialisiert werden. Auf der rechten Seite einer Zuweisung darf auch ein Objekt vom Typ `ADTValue<T2>` stehen, sofern `T2` $<:_{ADT}$ `T1` gilt.

Die Operation `ADT_cast<T1,T2>::apply` prüft diese Bedingung ab und löst einen Übersetzungsfehler aus, falls sie nicht zutrifft. Ansonsten liefert sie einen Zeiger auf die Konstruktorzelle der rechten Seite.

Mit `ADTValue` können wir nun festlegen, wie die Komponenten in einem Konstruktor gespeichert werden. Entspricht der Typ einer Komponente einem algebraischen Datentyp (bzw. einem Typkonstruktor), so speichern wir sie als Instanz von `ADTValue`, andernfalls als `EMLClosure`, da die Komponenten eines Konstruktors der verzögerten Auswertung unterliegen.

Jetzt fehlen nur noch die Operationen `pack` und `unpack`, um mit algebraischen Datentypen in C++ arbeiten zu können. Abbildung 11.5 zeigt beispielhaft deren Implementierung.

`pack` übernimmt den Typ eines mit `TypeInstantiate` instanziierten Konstruktors und stellt in `apply` eine Funktion zur Erzeugung eines neuen `ADTValue`-Objekts bereit.

11.5 Integration in EML

Beim Entwurf der Klassen `ADTValue` und der Wert- und Typkonstruktorklassen haben wir die verzögerte Auswertungsstrategie von EML bereits berücksichtigt, so daß wir sie zur Einbettung der algebraischen Typen in EML direkt wiederverwenden können.

Wir müssen daher nur noch die Operationen `pack`, `unpack` und `is` in EML bereitstellen. Dabei müssen wir drei Dinge berücksichtigen:

- Wertkonstruktoren dürfen auch partiell appliziert werden und
- ein Konstruktoraufruf kann zu einer Instanz eines polymorphen Typs gehören;
- Eine Konstruktorkomponente kann an einen Funktionsabschluß gebunden werden.

```
template <typename TYPEDESC>
class ADTValue {
private:
  ConsCell* value;
public:
  ADTValue(ConsCell* v) : value(v) { value->attach(); }
  ADTValue(const ADTValue& rhs) : value(rhs.value) {
    value->attach;
  }
  ~ADTValue() { value->detach() }

  ADTValue& operator=(const ADTValue& rhs) {
    rhs.value->attach();
    value->detach();
    value=rhs.value;
    return *this;
  }

  // Initialisierung mit und Zuweisung von einer
  // polymorphen Instanz T des algebraischen Typs
  // TYPEDESC
  template <typename T>
  ADTValue(const ADTValue<T>& v) : v(ADT_cast<TYPEDESC,T>::apply(v)) {
    v->attach();
  }
  template <typename T>
  ADTValue& operator=(const ADTValue<T>& rhs) {
    rhs.value->attach();
    value->detach();
    value=ADT_cast<TYPEDESC,T>::apply(rhs.value);
    return *this;
  }

  ConsCell* getRef() const { return value->getRef(); }
  int getId() const { return value->getRef()->getId(); }
};
```

Abbildung 11.4: Klassentemplate `ADTValue` zur Darstellung der Werte eines algebraischen Datentyps mit Typdeskriptor `TYPEDESC`.

```
template <typename TYPEDESC, int CONSID,
          typename A0,typename A1>
struct pack< ValueCons2<TYPEDESC,A0,A2> > {
  typedef ADTValue<TYPEDESC> RET;

  typedef typename mkConsArg<A0>::RET C0;
  typedef typename mkConsArg<A1>::RET B1;

  static inline RET apply(const C0& c0,const C1& c1) {
    return RET( new ValueCons2<TYPEDESC,A0,A1>( c0, c1) );
  }
};

template <typename TYPEDESC, int CONSID,
          typename A0,typename A1>
struct unpack< ValueCons2<TYPEDESC,A0,A1>, 0> {
  typedef A0 RET;
  typedef ValueCons2<TYPEDESC,A0,A1> Cons_t;
  static inline RET apply(const ADTValue<TYPEDESC>& v) {
#ifdef EML_Check_ConsAccess
    if (v->getId() != Cons_t::consId)
      throw IllegalConstrucorAcces();
#endif
    return ((Cons_t*)(v.value->getRef)).c0;
  }
};
```

Abbildung 11.5: Beispiele zur Implementierung von `pack` und `unpack`.

Der dritte Punkt zieht bezüglich der Instanziierung von Superkombinatoren gewisse Konsequenzen nach sich.

Nehmen wir an, es liegt ein Applikationsknoten auf dem Stack, dessen Auswertung zu einem Wert vom Typ `ADTValue<T>` führt. Vor der Instanziierung eines Superkombinators wird daraus (Anwendung von `mkGraphNode`) ein Wert vom Typ `EMLClosure<ADTValue<T> >` – der Applikationsknoten wurde in eine Heapzelle vom Typ `ClosureTerm` verschoben.

Nehmen wir weiter an, im Rumpf des Superkombinators befindet sich ein Aufruf von `pack`, der nun ein Objekt vom Typ `EMLClosure<ADTValue<T> >` erhält. Was soll `pack` mit diesem Knoten anfangen? Ihn auszuwerten kommt nicht in Frage, da wir damit gegen die Strategie der verzögerten Auswertung verstoßen würden. Wäre der Zeiger auf den Funktionsabschluß mit dem Interface von `ConsCell` kompatibel, könnten wir ihn unmittelbar in ein `ADTValue`-Objekt bewegen.

Wir spezialisieren `mkGraphNode` daher für den Fall, daß ein Term zu einem Wert mit algebraischem Typ führt und generieren einen speziellen Funktionsabschluß vom Typ `CCExpression`, der kompatibel zur Klasse `ConsCell` ist.

```
template <typename SCRIPT,typename E,typename R>
class CCexpression : public ConsCell< R > {
  char buffer[ sizeof(R) > sizeof(E) ? sizeof(E) : sizeof(R) ];
  CCExpression(const E& e) : ConsCell<R>( -2 ) {
    new (buffer) E(e);
  }
  ~CCExpression() {
    if (consId != -2)
      reinterpret_cast<E*>(buffer)->~E();
    else
      reinterpret_cast<R*>(buffer)->~R();
  }
  void run() const {
    emlStack.push( *reinterpret_cast<E*>( buffer ) );
    reinterpret_cast<E*>(buffer)->~E();
    new (buffer) R( translateCoreEML<SCRIPT,TML::C<E,TML::N> >::apply() );
    consId = reinterpret_cast<R*>(buffer)->getId();
  }
  ConsCell<void*> getRef() const {
    if (consId == -2) run();
    return buffer;
  }
  R operator()() const {
    if (consId == -2) run();
    return *reinterpret_cast<R*>(buffer);
  }
};
```

Abbildung 11.6: Hepzelle zur Aufnahme von nicht ausgewerteten Konstruktorkomponenten.

Das ist aber nicht so einfach, denn es ergibt sich folgendes *Design*-Problem: `CCExpression` muß alle Eigenschaften von `ConsCell` erben, sich aber nach wie vor wir ein Funktionsabschluß

verhalten, d.h. von `Closure<T>` erben. Mehrfachvererbung scheidet aus, da sowohl `ConsCell`, als auch `Closure<T>` von der Klasse `HeapCell` abgeleitet wurden. `CCExpression` würde daher über zwei Pfade von der Klasse `HeapCell` abgeleitet und das ist in C++ nicht erlaubt.

Wir lösen das Problem wie folgt (siehe Abbildung 11.6): Wir machen aus `ConsCell` ein Klassentemplate, daß wir von `Closure` ableiten und überschreiben den virtuellen `operator()()` so, daß er eine Exception auslöst, wenn er aufgerufen wird – im allgemeinen sind Konstruktorzellen ja keine Funktionsabschlüsse[3]. Die Klasse `CCExpression` kann dann von `ConsCell` erben und sich wie gewünscht als Konstruktorzelle und als Funktionsabschluß darstellen.

11.5.1 Syntaktische Integration

Betrachten wir zunächst ein Beispiel zur Demonstration des Ergebnisses der syntaktischen Integration:

```
( head(x)      = unpackCons<C,0>::apply(x),
  tail(x)      = unpackCons<C,1>::apply(x),
  cons(x)(y)   = packCons<N>::apply(x)(y) ,
  emptyList(x) = isCons<N>::apply(),
  length(x)    = If( emptyList(x),
                     0,
                     1 + tail(x)
                   )
)
```

Im Unterschied zu den C++ -Varianten, bekommen die Funktionen `packCons`, `unpackCons` und `isCons` jeweils die polymorphen Versionen der Wertkonstruktordeskriptoren übergeben. Zugegebenermaßen ist die Syntax sehr kryptisch, aber leider läßt uns C++ keine andere Wahl, statische und dynamische Information voneinander zu trennen. Als Ausweg bleibt aber, innerhalb von EML benutzerfreundlichere Funktionen zu definieren, so wie wir es im obigen Beispiel mit `cons`, `head`, `tail` und `emptyList` getan haben[4].

Wir beschränken uns darauf, die syntaktische Integration am Beispiel der Funktion `packCons` zu erläutern. Die beiden anderen Funktionen lassen sich nahezu analog realisieren.

[3] In der Tat wird diese Methode niemals aufgerufen, so daß es auch ausgereicht hätte, sie nur zu deklarieren. Würde sie dann dennoch aufgerufen, kommt es zu einem Fehler beim Binden des Programmes.

[4] Diese und andere Standardlistenoperationen sind in EML bereits vordefiniert.

```
typedef Var<-302> OpPack;

template <typename CONSDESC>
struct packCons {
static inline App<OpPack,packCons> apply() {
   return App<OpPack,packCons>(OpPack(), packCons() );
}
template <typename T>
static inline App<App<OpPack,packCons>,T> apply(const T& t) {
   return App<App<OpPack,packCons>,T>(
                App<OpPack,packCons>(OpPack(),packCons() ),
                t );
}
};
```

Zwei Varianten von `apply` werden zur Verfügung gestellt. Die erste dient der Erzeugung von nullstelligen Konstruktoren, mit der zweiten können mehrstellige Konstruktoren erzeugt werden. Zur Erinnerung: Applikationsknoten stellen einen Funktionsaufrufoperator zur Verfügung, so daß das Ergebnis der Anwendung von `pack` auf weitere Argumente angewendet werden kann.

Im Applikationsknoten hinterlegen wir nicht die Wertkonstruktorklasse, sondern einen Verweis auf die Klasse `packCons`. Damit vermeiden wir, daß im Applikationsknoten Speicherplatz für eine Instanz der Wertkonstruktorklasse reserviert wird. Wir brauchen ja nur dessen Typ und den können wir durch den *pattern matching*-Mechanismus der Template-Spezialisierung jederzeit extrahieren.

11.5.2 Erweiterung der Typberechnung

Die Typinferenz für `isCons` und `unpackCons` läßt sich sehr einfach realisieren. `isCons` generiert grundsätzlich ein Ergebnis vom Typ `bool`.

`unpackCons<ConsDesc,N>` kann nur auf Werte vom Typ `ADTValue<TypeDesc>` bzw. `EMLClosure< ADTValue<TypeDesc> >` angewendet werden. Die im Typdeskriptor hinterlegten Typargumente nutzen wir, um den polymorphen Konstruktordeskriptor mittels `TypeInstantiate` zu instanziieren. Dabei muß ein monomorpher Wert entstehen, ansonsten liegt ein Typfehler vor. Das Ergebnis der Typberechnung entspricht dann der Projektion auf die `N`-te Komponente des instanziierten Konstruktortyps.

Bei der Anwendung von `packCons<ConsDesc>` kann eine polymorphe Instanz des Typs entstehen, zu dem der Konstruktordeskriptor `ConsDec` gehört. Mit Hilfe der `typeId` und der Klasse `DefineSumType` können sowohl der generische Konstruktortyp, als auch der generische Typdeskriptor des algebraischen Typs generiert werden. Zusammen mit dem Typ der an `packCons<ConsDesc>` übergebenen Argumente läßt sich so sehr leicht eine eventuell polymorphe Instanz `TypeDescInst` des Typdeskriptors berechnen. Der Ergebnistyp von `packCons` ist dann ein Wert dieses Typs, also eine Instanz von `ADTValue<TypeDescInst>`.

Polymorphe Instanzen müssen aber auch bei der Typisierung der einfachen Selektion (`if`-Terme) berücksichtigt werden. Führen then- und else-Zweig zu unterschiedlichen Instanzen,

berechnen wir den Ergebnistyp wie folgt. Seien `ADTValue` < `TypeCons` $<N, a_1, \cdots, a_n> >$ und `ADTValue` < `TypeCons` $<M, b_1, \cdots, b_m> >$ die Ergebnistypen der beiden Zweige. Dann muß gelten:

- $N = M$ und $n = m$ – die resultierenden Werte müssen demselben algebraischen Typ angehören.
- Zwei Typargumente a_i und b_i müssen exakt übereinstimmen[5], oder eines der Typargumente ist eine Typvariable.

Sind diese Bedingungen erfüllt, erhalten wir den Ergebnistyp, indem wir die Typargumente vergleichen und jeweils das Element selektieren, welches keine Typvariable ist.

11.5.3 Implementierung der Transitionsschritte

Bei der Codegenerierung für `isCons` und `unpackCons` kann auf die entsprechenden Routinen `is` und `unpack` zurückgegriffen werden. Liegt ein unausgewerteter Knoten als Argument vor (`EMLClosure<ADTValue<T> >`), so muß dieser durch Aufruf von `operator()()` zunächst ausgewertet werden.

Die Implementierung von `packCons` ist ein wenig aufwendiger. Wie bei der Umsetzung von Grundoperationen, erkennen wir Anwendungen von `OpPack`, bevor die Argumente auf den Stack verschoben werden. Es kann sich also um Terme oder Funktionsabschlüsse handeln. Bevor wir diese an `pack` weitergeben, müssen sie entsprechend angepaßt werden:

- Liegt auf dem Stack eine C++ -Konstante, müssen wir einen Funktionsabschluss vom Typ `ClosureConstant` erzeugen und in ein `EMLClosure`-Objekt einschließen.
- Liegt ein Applikationsknoten auf dem Stack, so erzeugen wir ein `ClosureExpression`-Objekt, falls die zugehörige Konstruktorkomponente einen Wert vom Typ `EMLClosure` erwartet, oder ein `CCExpression`-Objekt, wenn in der Komponente ein algebraischer Wert abgelegt werden soll.
- Liegt ein Objekt vom Typ `EMLClosure<ADTValue<T> >` auf dem Stack, wir erwarten aber einen Wert vom Typ `ADTValue<T>`, so kann man Gewiß sein, daß der in `EMLClosure` gekapselte Zeiger auf eine Heapzelle vom Typ `CCExpression` zeigt – dafür hatten wir mit der Spezialisierung der Klasse `mkGraphNode` ja Sorge getragen. Wir können diesen Zeiger somit unmittelbar zur Konstruktion eines `ADTValue`-Objekts einsetzen.

11.6 Ergebnisse

Zum Test, wie konkurrenzfähig unser *staged interpreter* im Vergleich zu speziellen Übersetzern für funktionale Sprachen ist, greifen wir auf die Berechnung der Primzahlen mit dem Sieb des Erathostenes zurück. Dieser Algorithmus bietet sich an, da er Gebrauch von unendlichen Listen macht und der Heap durch ständig erzeugte und freigegebene Objekte stark belastet wird.

[5] Man könnte auch fordern, daß sie ein Supremum bzgl. der Untertyprelation <: besitzen. Das wäre ein wenig flexibler.

Es sei vorweggenommen, daß wir keine Wunder erwarten dürfen, da wir keinerlei *high-level* Optimierungen vornehmen. Abgesehen von der Elimination von endrekursiven Aufrufen, beziehen sich alle von uns vorgenommenen Optimierungen auf die virtuelle Maschine, für die wir übersetzen. Es erscheint daher fair, die professionellen Übersetzer ohne eingeschaltete Optimierung antreten zu lassen.

```
EMLList<int> primes;
int count;

cout << "Number of primes to compute: " << flush;
cin >> count;

primes = runEML(
         (  import(head), import(tail),
            import(isNIL),import(cons),
            import(nil),  import(filter),
            import(take),

            from(!x)          = cons(x)(from(x+1)),
            fromTo(!x)(!y)    = IF(x>y,
                                   nil(),
                                   cons(x) ( fromTo(x+1)(y) )
                                  ),
            divisible(!x)(!y) = (x % y) == 0,

            factors(!x)       = filter(divisible(x))(fromTo(1)(x)),
            isPrime(!x)       = factors(x) == (cons(1)(cons(x)(nil()))),

            m(x)              = take(x)( filter(isPrime)(from(1)) )
          ),
        m(count)
       );

cout << endl << "primes=" << flush << primes << endl;

cout << "This time it should be significantly faster:" << endl << endl;
cout << endl << "primes=" << flush << primes << endl;
```

Abbildung 11.7: Primzahlberechnung in EML.

Abbildung 11.7 zeigt das EML-Skript zur Berechnung der Primzahlen. Die Klasse `EMLList` kapselt ein Objekt vom Typ `ADTValue` und stellt einige Methoden zur Verfügung, so daß der Programmierer Listen von algebraischem Typ verwenden kann, als ob es sich um eine normale C++ -Containerklasse handeln würde. Insbesondere wird auch ein Ausgabeoperator (`operator<<`) bereitgestellt.

Bei `import` handelt es sich um einfaches Makro, das eine der vordefinierten Listenfunktionen in das aktuelle Skript importiert.

Wie Abbildung 11.8 zeigt, erzeugt unser *staged interpreter* deutlich schnelleren Programmcode als der Haskell Interpreter ghci und der Haskell Übersetzer nhc98 (bis zu Faktor 8). Den

# Primzahlen	EML (g++)	ghc	ghc -O	ghci	nhc98
500	0.4s	0.3s	0.1s	3.3s	3.1s
1000	1.9s	1.6s	0.8s	15.5s	13.4s
1500	5.1s	3.2s	1.7s	35.0s	31.9s
2000	10.0s	6.5s	3.5s	68.9s	58.9s

Abbildung 11.8: Laufzeitvergleich für das Primzahl-Testprogramm auf unserem Pentium 4M Testsystem. Der Haskell Übersetzer nhc98 wurde in der Version 1.16 verwendet.

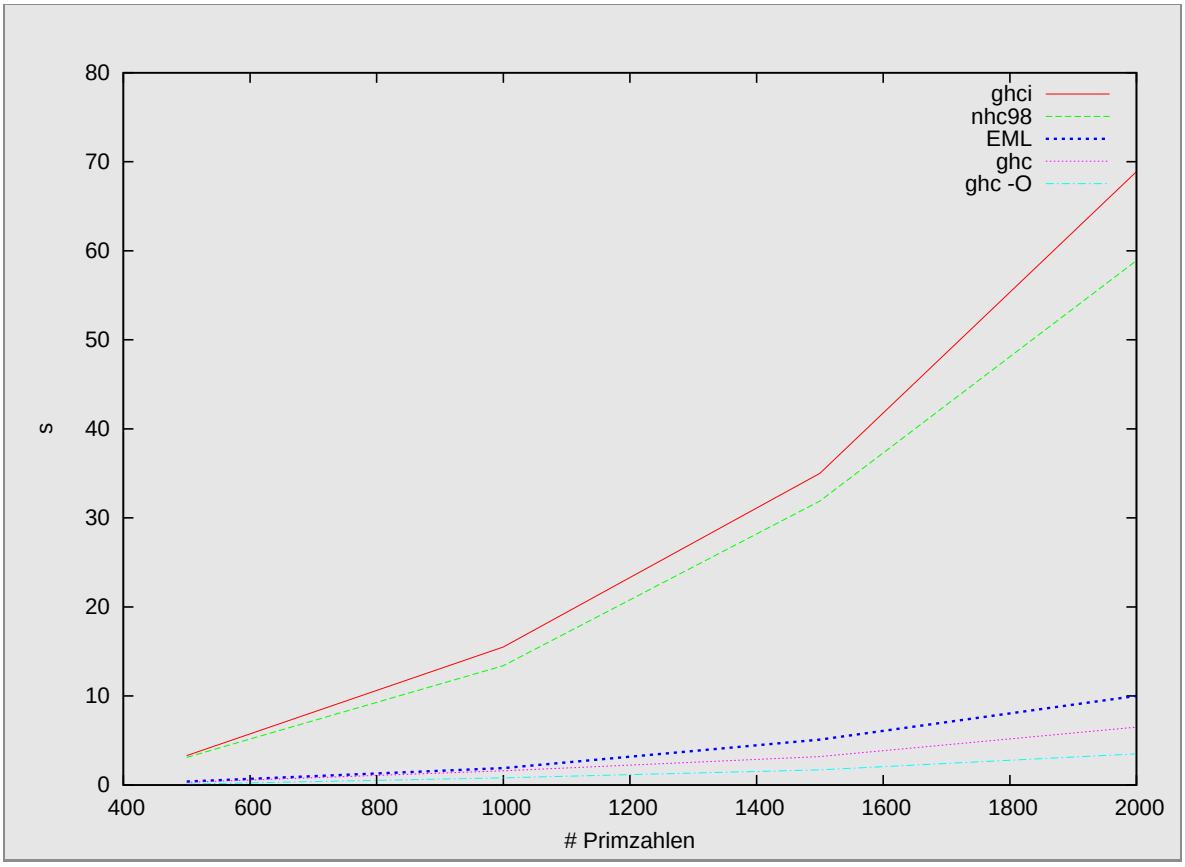


Abbildung 11.9: Grafische Darstellung zu Abbildung 11.8.

Vergleich mit dem Glasgow Haskell Compiler brauchen wir ebenfalls nicht zu scheuen, denn von ihm trennt uns lediglich ein Faktor von 1.1 - 1.5.

Erlaubt man dem Glasgow Haskell Compiler jedoch Optimierungen vorzunehmen, steigt der Faktor auf 2.3 - 2.8. Das ist nicht weiter verwunderlich, da ghc bei eingeschalteter Optimierung eine ausgedehnte Striktheitsanalyse durchführt, die Konstruktoren einschließt. Bei der Berechnung der Primzahlen kann das Kopfelement der Liste jeweils strikt ausgewertet werden, was die Last auf dem Heap enorm reduziert (siehe [174]).

Nicht zu vernachlässigen ist aber auch, daß die *garbage collection* via Referenzzählung im Vergleich zu anderen Verfahren (z.B. der in Glasgow Haskell Compiler eingesetzten *generational garbage collection* – siehe [89, S. 143 ff]) deutlich ineffizienter ist.

Auch wenn die Laufzeiten sehr vielversprechend aussehen, ist der Einsatz von EML in Softwareprojekten nicht zu empfehlen, ja sogar ausgeschlossen. Wie wir am Ende von Kapitel 5 bereits erwähnt haben, ist der C++ -Template-Mechanismus weder hinsichtlich des Speicherverbrauchs, noch hinsichtlich der Laufzeit sonderlich effizient. So dauerte die Übersetzung des Primzahlprogrammes auf unserem Testsystem (Pentium 4M) etwa vier Minuten und benötigte ungefähr dreihundert Megabyte Arbeitsspeicher – das ist untragbar!

Aber es ging uns ja weniger darum, eine alltagsfähige Multiparadigma-Sprache zu generieren. Vielmehr wollten wir demonstrieren, daß strukturelle Typanalyse in Verbindung mit partieller Auswertung ein brauchbares Werkzeug zur Sprachintegration darstellt. Das wurde mit Rückblick auf die Ergebnisse, die wir bei der Optimierung von endrekursiven Aufrufen und gerade bei der Primzahlberechnung erhalten haben, unter Beweis gestellt.

Kapitel 12

Zusammenfassung und Ausblick

12.1 Zusammenfassung

Im ersten Teil dieser Arbeit haben wir die Modellsprache System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ zur Programmierung mit struktureller Typanalyse entwickelt. Am Beispiel von Überladung und *pattern matching* haben wir gezeigt, wie man polymorphe Funktionen auf bestimmte Typen einschränken kann und den Unterschied zu beschränkter parametrischer Polymorphie erklärt.

Durch die Hinzunahme eines partiellen Auswerters zur Sprachdefinition von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$, haben wir aus System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ eine zweistufige Sprache gemacht. In der ersten Stufe werden typabhängige Terme reduziert und es wird Programmcode für die zweite Stufe erzeugt. Der Kontrollfluß des entstandenen Programmes ist unabhängig von der Typinformation, so daß diese vor der Übersetzung in Maschinencode eliminiert werden kann. Insgesamt sind Typberechnungen und Wertberechnungen dadurch klar voneinander abgegrenzt.

Die Zweistufigkeit von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ haben wir dann genutzt, um einen *staged interpreter* für eine einfache, in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ eingebettete Sprache zu realisieren. Wir haben gezeigt, wie man mit Hilfe der strukturellen Typanalyse ein Typsystem für eingebettete Sprachen programmiert, so daß Typfehler in der Gastsprache bereits zur Übersetzungszeit der Hostsprache angezeigt werden.

Durch die Verlagerung der *case analysis* in die Übersetzungszeit haben wir erreicht, daß der Aufwand zur Interpretation der Gastsprache (z.B. *case analysis*, Traversieren des AST) zur Übersetzungszeit der Hostsprache stattfindet. Effektiv ist der *staged interpreter* damit zu einem Übersetzer von der Gast- in die Hostsprache geworden (Futamura Projektion).

An zwei einfachen Beispielen haben wir gezeigt, daß sich der *staged interpreter* Ansatz mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ auch zur Implementierung von typabhängigen (domänenspezifischen) Optimierungen eignet, so daß man einen System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$-Übersetzer gewissermaßen um Optimierungsphasen erweitern kann, ohne den Übersetzer selbst ändern zu müssen.

Zur Einbettung von komplexeren Sprachen (mit Rekursion und komplexeren Typsystemen) haben wir zwei Erweiterungen von System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ vorgestellt. Wir mußten feststellen, daß diese Erweiterungen jedoch negative Auswirkungen auf die Eigenschaften der erweiterten Sprache haben. In System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ können zwar rekursive (berechnungsuniverselle) Sprachen eingebettet werden, allerdings ist die Terminierung des partiellen Auswerters nicht mehr garantiert. Sprachen, die ein komplexes Typsystem mitbringen, dessen Implementierung selbst nach einer berechnungsuniversellen Sprache verlangt, lassen sich mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ einbetten. Da wir

das Typsystem der eingebetteten Sprache innerhalb des Typsystems von System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ kodieren müssen, verlieren wir mit System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ jedoch die Entscheidbarkeit der Typisierung eines Terms.

Zusammenfassend bietet unser neuer Ansatz folgende Vorteile

- Beim Erstellen von Übersetzern für domänenspezifische Sprachen (oder allgemein: der Einbettung einer Gast- in eine Hostsprache) können die syntaktische Analysephase, die Optimierungsphase und die Codegenerierungsphase eines bestehenden Übersetzers wiederverwendet werden.

- Für die einzubettende Sprache sind im wesentlichen ein Interpreter und ein *Typechecker* zu implementieren – softwaretechnisch gesehen ist der dazu notwendige Aufwand deutlich geringer, als die Programmierung eines Übersetzers.

- Durch partielle Auswertung erzielt man Laufzeitqualitäten, die man normalerweise nur mit Übersetzern erreichen kann.

- Im Vergleich zu anderen Ansätzen zur Typanalyse von eingebetteten Sprachen bietet unser Ansatz Vorteile. Im Vergleich zu *phantom types* können wir zur Typanalyse auf eine berechnungsuniverselle Sprache zurückgreifen. Außerdem können Programme klarer formuliert werden, da bestehende Sprachkonzepte nicht mißbraucht werden (so wie die Typklassen im Fall der *phantom types*), sondern mit `Typecase` und `typecase` explizit zur Verfügung stehen. Überladung durch ein Termersetzungssystem zu steuern, macht den Programmcode zwar besser lesbar, hat aber zur Folge, daß die Entscheidbarkeit der Typzuordnung zu einem Term verloren geht. Dies ist insofern fatal, als daß der Übersetzer ggf. nicht terminiert.

  Um nicht auf die Entscheidbarkeit der Typisierung verzichten zu müssen, erlaubt die hier vorgestellte System $\mathbf{F}^{\mathrm{SA}}_{\omega}$-Sprachfamilie dem Programmierer, sich auf die Sprachmittel zu beschränken, die er tatsächlich braucht.

- Unser Ansatz ist nicht auf einen algebraischen Datentyp beschränkt, sondern kann auf beliebige Typen ausgedehnt werden. Dadurch besteht die Möglichkeit, bestehende Programme um domänenspezifische Optimierungen zu erweitern.

Unsere ersten beiden Hypothesen haben sich bestätigt:

H1 Strukturelle Typanalyse, gepaart mit partieller Auswertung, ist ein geeignetes Werkzeug zur Einbettung einer getypten Gastsprache in eine Hostsprache.

Wir konnten an zahlreichen kleinen Beispielen im ersten Teil und anhand des zweiten Teils der Arbeit nachweisen, daß Spracheinbettung mit struktureller Typanalyse möglich ist. Es hat sich gezeigt, daß wir auch Sprachen mit einem von der Hostsprache abweichenden Typsystem einbetten können.

H2 Diese neue Technik der Spracheinbettung erlaubt prinzipiell die Übersetzung der Gastsprache in laufzeiteffizienten Maschinencode.

Im theoretischen Teil haben wir nachgewiesen, daß der typische Interpretationsaufwand durch partielle Auswertung zu einem Großteil in die Übersetzungszeit verlagert werden

kann. Diese Aussage sehen wir durch die Ergebnisse des Praxisteils bestätigt. Insbesondere konnten wir im Praxisteil am Beispiel der Optimierung von endrekursiven Aufrufen nachweisen, daß unser Ansatz sich dazu eignet, Optimierungen für die Gastsprache zu vereinbaren, die der Übersetzer der Hostsprache nicht unterstützt.

In Abschnitt 5.3 haben wir gezeigt, daß sich viele Konzepte aus System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ in der Programmiersprache C++ simulieren lassen. Andersherum betrachtet, bietet sich System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ als erstes Modell zur Beschreibung der Phänomene der C++ -Template-Metaprogrammierung an, wodurch wie unsere dritte Hypothese bestätigt sehen.

H3 Unsere Methode ist geeignet, die Phänomene der C++ -Template-Metaprogrammierung zu beschreiben.

12.2 Offene Fragestellungen und mögliche Anschlußarbeiten

Die Zweistufigkeit der Modellsprachen System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ bis System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ ergibt sich aus der strikten Trennung von Typüberprüfung und Termauswertung. Die separate Übersetzung von Programmmodulen wird dadurch allerdings schwer, denn typabhängige Terme müssen zur Übersetzungszeit in Quelltextform verfügbar sein.

Zwar stellt Weirich in [39] ein auf System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$ anwendbares Verfahren vor, welches die Übersetzung mittels Typelimination ermöglichen würde, jedoch müssen dabei einige typabhängige Terme zur Laufzeit ausgewertet werden, womit die für uns wichtige Zweistufigkeit verloren geht.

Bei der Implementierung von EML haben wir an vielen Stellen gesehen, daß ein erweiterbares *pattern matching*, so wie es der C++ -Template-Mechanismus zur Verfügung stellt, ein enorm mächtiges Werkzeug ist, um nachträglich Erweiterungen an der eingebetteten Sprache vorzunehmen. Im Vergleich dazu ist das *pattern matching* in den Sprachen der System $\mathbf{F}^{\mathrm{SA}}_{\omega}$-Familie abgeschlossen (es können keine Behandlungsfälle hinzugefügt werden, ohne die entsprechende Funktion zu ändern und es wird immer das erste passende Muster gewählt). An dieser Stelle stimmt das Modell nicht mit der C++ -Template-Metaprogrammierung überein. Es stellt sich die Frage, ob man System $\mathbf{F}^{\mathrm{SA}}_{\omega}$-Sprachen entsprechend erweitern kann – insbesondere vor dem Hintergrund der separaten Übersetzung.

Eine Frage, die wir im Rahmen dieser Arbeit nahezu unbeantwortet gelassen haben, und die man im Allgemeinen auch nur schwer beantworten kann, ist die, ob der partielle Auswerter garantiert sämtlichen „Interpretationsoverhead" entfernt (siehe auch [84, Problem 3.8]). Intuitiv gesprochen verbirgt sich dahinter die Frage, ob das vom partiellen Auswerter generierte Residuum genauso effizient ist, wie Programmcode, den ein Programmierer, der sich mit den Eigenarten der Gastsprache auskennt, von Hand generiert hätte.

Ein gebräuchliches Kriterium zur Beurteilung der Qualität eines partiellen Auswerters (bezüglich der Spezialisierung von Interpretern), ist die **Jones-Optimalität** eines Selbstinterpreters [160][108]. Ein Selbstinterpreter ist ein Interpreter, der in derselben Programmiersprache implementiert ist, die er interpretiert. Stimmen zu interpretierender Code und das Residuum überein, ist der partielle Auswerter Jones-optimal [108]. Allgemeiner kann man auch sagen, daß ein partieller Auswerter Jones-optimal ist, wenn das Residuum des Selbstinterpreters für beliebige Eingabewerte mindestens genauso Laufzeiteffizient ist, als wenn man den zu interpretierenden Programmtext direkt codiert hätte (siehe [88, Abschnitt 5.4]).

Eine interessante Frage wäre, ob sich in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$, System $\mathbf{F}^{\mathrm{SA}}_{\omega,2}$ und System $\mathbf{F}^{\mathrm{SA}}_{\omega,3}$ überhaupt ein Selbstinterpreter realisieren läßt, und ob dieser das Kriterium der Jones-Optimalität erfüllt, bzw. ob und wie diese Sprachen entsprechend erweiterbar sind.

Interessant wäre auch, zu untersuchen, inwieweit sich die Typsysteme von etablierten Programmiersprachen (z.B. JAVA, C und FORTRAN) um das Konzept der strukturellen Typanalyse erweitern lassen.

Bezogen auf den zweiten Teil der Arbeit (insbesondere die Template-Metaprogrammierung), ergeben sich zahlreiche Problemstellungen, zu denen man in Anschlussarbeiten Lösungen suchen könnte. Ein großes Problem der Template-Metaprogrammierung ist der Speicherverbrauch zur Übersetzungszeit. Man könnte z.B. untersuchen, ob sich die Sprache C++ so erweitern ließe, daß man Metafunktionen mit einem speziellen Attribut (z.B. `dropInstanve`) versehen kann, um den Übersetzer anzuweisen, eine Instanz dieses Templates sofort wieder zu verwerfen, wenn das Ergebnis der Metaberechnung ermittelt wurde. Viel Optimierungspotential tut sich aber auch bei der Ausführung von Template-Metaprogrammen auf.

Anhang A

Im Text verkürzt dargestellte Herleitungen

A.1 Partielle Auswertung von $\mathtt{eq}_P\,[\![(7,9)]\!]$

Aus Platzgründen zeigen wir nur die Schritte von $\mathcal{T}$, die eine Veränderung bewirken. Zunächst wird die $\mathcal{T}$-Regel zur Behandlung von Überladungsapplikationen angewendet.

$$\mathtt{eq}_P\,[\![(7,9)]\!] = \mathcal{L}[\![\ \mathcal{T}[\![\mathtt{eq}_P[\times(\mathtt{int},\mathtt{int})]]\!]\ \ \mathcal{T}[\![\times(7,9)]\!]\ ]\!]$$

Im nächsten Schritt gilt es $\mathcal{T}[\![\mathtt{eq}_P[\times(\mathtt{int},\mathtt{int})]]\!]$ zu reduzieren. Typapplikationen werden direkt ausgewertet:

$$\mathcal{T}[\![\mathtt{eq}_P[\times(\mathtt{int},\mathtt{int})]]\!] = \mathcal{T}[\![\mathtt{typecase}\ \times(\mathtt{int},\mathtt{int})\ \mathtt{of}\ \cdots]\!]$$

Diese `typecase`-Anweisung wird durch partielles Auswerten von **E-TCase-Cons-2** reduziert:

$$\begin{aligned}\mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![\ \mathcal{T}[\![\ & (\Lambda\alpha :: *.\lambda f_0 : \alpha \to \mathtt{bool}.\ \ \Lambda\beta :: *.\lambda f_1 : \beta \to \mathtt{bool}.\\ & \lambda c : \times(\alpha,\beta).\mathtt{eq}[\alpha]\ \ c.0\ \ c.1)[\mathtt{int}]]\!]\\ & (\mathcal{T}[\![\mathtt{typecase}\ \mathtt{int}\cdots(*\mathit{entsprechend}\ \mathtt{eq}_P[\mathtt{int}]*)]\!])]\!]\\ & [\mathtt{int}]]\!](\mathcal{T}[\![\mathtt{typecase}\ \mathtt{int}\cdots(*\mathit{entsprechend}\ \mathtt{eq}_P[\mathtt{int}]*)]\!])]\!]\end{aligned}$$

Durch Ausrechnen der Typ-Typ-Applikation gelangt man zu:

$$\begin{aligned}\mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![(\lambda f_0 : \mathtt{int} \to \mathtt{bool}.\Lambda\beta :: *.\lambda f_1 : \beta \to \mathtt{bool}.\\ & \lambda c : \times(\mathtt{int},\beta).\ \mathcal{T}[\![\mathtt{eq}[\mathtt{int}]]\!]\ \ c.0\ \ c.1)\\ & (\mathcal{T}[\![\mathtt{typecase}\ \mathtt{int}\cdots(*\mathit{entsprechend}\ \mathtt{eq}_P*)]\!])]\!]\\ & [\mathtt{int}]]\!](\mathcal{T}[\![\mathtt{typecase}\ \mathtt{int}\cdots(*\mathit{entsprechend}\ \mathtt{eq}_P[\mathtt{int}]*)]\!])]\!]\end{aligned}$$

Wir man leicht nachvollzieht ist $\mathtt{eq}[\mathtt{int}] = ==_{\mathtt{int}}$:

$$\begin{aligned}&\mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![(\lambda f_0 : \texttt{int} \to \texttt{bool}.\Lambda\beta :: *.\lambda f_1 : \beta \to \texttt{bool}.\\ &\qquad \lambda c : \times(\texttt{int}, \beta).==_{\texttt{int}}\ c.0\ \ c.1)\\ &\qquad (\ \mathcal{T}[\![\texttt{typecase int} \cdots (*\mathit{entsprechend}\ \texttt{eq}_P[\texttt{int}]*)]\!]\ )]\!]\\ &\qquad [\texttt{int}]]\!](\mathcal{T}[\![\texttt{typecase int} \cdots (*\mathit{entsprechend}\ \texttt{eq}_P[\texttt{int}]*)]\!])]\!]\end{aligned}$$

Die jetzt zu generierenden Funktionen f_0 und f_1 sind für das Ergebnis belanglos; wir fahren dennoch fort:

Mit $\texttt{eq}_P[\texttt{int}] = \lambda x : \texttt{int}.\texttt{false}$ ergibt sich:

$$\begin{aligned}&\mathcal{L}[\![\mathcal{T}[\![\ \mathcal{L}[\![\ (\lambda f_0 : \texttt{int} \to \texttt{bool}.\Lambda\beta :: *.\lambda f_1 : \beta \to \texttt{bool}.\\ &\qquad\qquad \lambda c : \times(\texttt{int}, \beta).==_{\texttt{int}}\ c.0\ \ c.1)\\ &\qquad\qquad (\lambda x : \texttt{int}.\texttt{false})]\!]\\ &\qquad\qquad [\texttt{int}]]\!](\mathcal{T}[\![\texttt{typecase int} \cdots (*\mathit{entsprechend}\ \texttt{eq}_P*)]\!])]\!]\end{aligned}$$

Mit einem β-Reduktionsschritt (Anwenden von $\mathcal{L}$) gelangt man zu:

$$\begin{aligned}&\mathcal{L}[\![\ \mathcal{T}[\![\ (\Lambda\beta :: *.\lambda f_1 : \beta \to \texttt{bool}.\\ &\qquad\qquad \lambda c : \times(\texttt{int}, \beta).==_{\texttt{int}}\ c.0\ \ c.1)\\ &\qquad\qquad [\texttt{int}]]\!]\ (\mathcal{T}[\![\texttt{typecase int} \cdots (*\mathit{entsptrechend}\ \texttt{eq}_P*)]\!])]\!]\end{aligned}$$

Ausrechnen der Typapplikation ergibt:

$$\begin{aligned}&\mathcal{L}[\![(\lambda f_1 : \texttt{int} \to \texttt{bool}.\\ &\quad \lambda c : \times(\texttt{int}, \texttt{int}).==_{\texttt{int}}\ c.0\ \ c.1)\\ &\quad (\ \mathcal{T}[\![\texttt{typecase int} \cdots (*\mathit{entsptrechend}\ \texttt{eq}_P*)]\!]\ )]\!]\end{aligned}$$

Mit $\texttt{eq}_P[\texttt{int}] = \lambda x : \texttt{int}.\texttt{false}$ folgt:

$$\begin{aligned}&\mathcal{L}[\![\ (\lambda f_1 : \texttt{int} \to \texttt{bool}.\\ &\qquad \lambda c : \times(\texttt{int}, \texttt{int}).==_{\texttt{int}}\ c.0\ \ c.1)\\ &\qquad (\lambda x : \texttt{int}.\texttt{false})]\!]\end{aligned}$$

Durch β-Reduktion gelangen wir zu:

$$\lambda c : \times(\texttt{int}, \texttt{int}).==_{\texttt{int}}\ c.0\ \ c.1$$

Einsetzen in Gleichung A.1 führt zu:

$$\mathcal{L}[\![(\lambda c : \times(\texttt{int}, \texttt{int}).==_{\texttt{int}}\ (c.0\ \ c.1)\ \times(7, 9)]\!]$$

Ausrechnen der Applikation (β-Reduktion) ergibt:

$$\mathcal{T}[\![==_{\texttt{int}}\ \times(7, 9).0\ \ \times(7, 9).1]\!]$$

Mit der Regel zur partiellen Auswertung von Applikatonen wandert $\mathcal{T}$ nach innen und wird auf die beiden Projektionen angewendet:

$$==_{\texttt{int}} \quad \mathcal{T}[\![\times(7,9).0]\!] \quad \mathcal{T}[\![\times(7,9).1\ ]\!]$$

Ausrechnen der Projektionen führt zum Ergebnis

$$==_{\texttt{int}} \ 7 \ 9$$

Die Überladung des Vergleichsoperators wird folglich zur Übersetzungszeit durch den partiellen Auswerter aufgelöst.

A.2 Parsing von add ⟦(x, mul ⟦(x, y)⟧)⟧

Abbildung A.1 zeigt eine im Vergleich zu Abbildung 4.4 auf Seite 88 effizientere Version eines Parsers für **V**. Die wesentliche Änderung liegt im Funktionstyp, der von $\texttt{check}_{\mathbb{V}}$ generierten Funktion. Als erstes Argument wird hier ein Funktionstyp erwartet. Dieses Argument wird jedoch nicht weiter verwendet, sondern dient in erster Linie dazu, anstelle von Funktionsapplikationen auf Überladungsapplikationen zurückgreifen zu können. Aus Sicht der partiellen Auswertung liegt der Unterschied darin, daß Funktionsapplikationen unangetastet bleiben, während für Überladungsapplikationen ein β-Reduktionsschritt vollzogen wird.

Die Herleitung ist relativ lang. Im Sinne einer übersichtlichen Darstellung leiten wir zunächst einige Teilterme her, auf die wir im weiteren Verlauf zurückgreifen werden.

$$\begin{aligned}
\mathcal{T}[\![\texttt{check}_{\mathbb{T}}[\texttt{String}]]\!] &= \mathcal{T}[\![\ \forall\gamma :: *.\texttt{marshall}_{\mathbb{T}}[\texttt{String}]\ ]\!] \\
&= \forall\gamma :: *.\texttt{void} \to \texttt{FALSE} \\
\mathcal{T}[\![\texttt{check}_{\mathbb{T}}[\texttt{Var(String)}]]\!] &= \forall\gamma :: *.\texttt{Var(String)} \to \texttt{Var(String)} \\
\mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{String}]]\!] &= \mathcal{T}[\![\ \Lambda\gamma :: *.\texttt{marshall}_{\mathbb{V}}[\texttt{String}]\ ]\!] \\
&= \Lambda\gamma :: *.\lambda x : \texttt{void}.\texttt{FALSE} \\
\mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{Var(String)}]]\!] &= [\![\ (\Lambda\delta :: *.\texttt{typecase}\ \delta\langle\Lambda\gamma :: *.\texttt{check}_{\mathbb{T}}[\gamma]\rangle\ \texttt{of}\ \cdots)[\texttt{Var(String)}]\ ]\!] \\
&= [\![\texttt{typecase}\ \texttt{Var(String)}\langle\Lambda\gamma :: *.\texttt{check}_{\mathbb{T}}[\gamma]\rangle\ \texttt{of}\ \cdots)]\!] \\
&= \mathcal{L}[\![\ \mathcal{T}[\![(\ \Lambda\alpha :: *.\lambda f : \texttt{check}_{\mathbb{T}}[\alpha].\ \\
&\qquad\qquad \Lambda\gamma' :: *.\lambda v : \texttt{Var}(\alpha).v)[\texttt{String}]]\!] \\
&\qquad (\mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{String}]]\!])]\!] \\
&= \mathcal{L}[\![\lambda f : \mathcal{T}[\![\texttt{check}_{\mathbb{T}}[\texttt{String}]]\!]\ . \\
&\qquad \Lambda\gamma' :: *.\lambda v : \texttt{Var}(\texttt{String}).v]\!] \\
&\qquad (\mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{String}]]\!])]\!] \\
&= \cdots
\end{aligned}$$

$$
\begin{array}{rcl}
\mathtt{Res}_{\mathbb{T}} & := & \Lambda\delta :: *.\mathtt{Typecase}\ \delta\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \Lambda\alpha :: *.\mathtt{FALSE} \\
& & \quad \to: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K.\beta \\
\mathtt{Arg}_{\mathbb{T}} & := & \Lambda\delta :: *.\mathtt{Typecase}\ \delta\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \Lambda\alpha :: *.\mathtt{FALSE} \\
& & \quad \to: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K.\alpha \\
\mathtt{marshall}_{\mathbb{T}} & := & \Lambda\delta :: *.\mathtt{Typecase}\ \delta\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \mathtt{void} \to \mathtt{FALSE} \\
& & \quad \mathtt{int}: \ \mathtt{int} \to \mathtt{Num}(\mathtt{int}) \\
\mathtt{marshall}_{\mathbb{V}} & := & \Lambda\delta :: *.\mathtt{typecase}\ \delta\ \ \langle\Lambda\gamma :: *.\mathtt{marshall}_{\mathbb{T}}[\gamma]\rangle\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \Lambda\alpha :: *.\lambda x : \mathtt{void}.\mathtt{FALSE} \\
& & \quad \mathtt{int}: \ \lambda x : \mathtt{int}.\mathtt{Num}(x) \\
\mathtt{check}_{\mathbb{T}} & := & \Lambda\delta :: *.\mathtt{Typecase}\ \delta\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \Lambda\alpha :: *.\Lambda\gamma :: *.\mathtt{marshall}_{\mathbb{T}}[\alpha] \\
& & \quad \mathtt{Var} \ \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\gamma :: *.\mathtt{Var}(\mathtt{String}) \to \mathtt{Var}(\mathtt{String}) \\
& & \quad \mathtt{Num}: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\gamma :: *.\mathtt{Num}(\mathtt{int}) \to \mathtt{Num}(\mathtt{int}) \\
& & \quad \mathtt{Add}: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K. \\
& & \qquad \Lambda\gamma :: *.\ \mathtt{Add}(\mathtt{Arg}_{\mathbb{T}}[\alpha'[\alpha]], \mathtt{Arg}_{\mathbb{T}}[\beta'[\eta]]) \to \\
& & \qquad\quad \mathtt{Add}(\mathtt{Res}_{\mathbb{T}}[\alpha'[\alpha]], \mathtt{Res}_{\mathbb{T}}[\beta'[\beta]]) \\
& & \quad \mathtt{Mul}: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K. \\
& & \qquad \Lambda\gamma :: *.\ \mathtt{Mul}(\mathtt{Arg}_{\mathbb{T}}[\alpha'[\alpha]], \mathtt{Arg}_{\mathbb{T}}[\beta'[\beta]]) \to \\
& & \qquad\quad \mathtt{Mul}(\mathtt{Res}_{\mathbb{T}}[\alpha'[\alpha]], \mathtt{Res}_{\mathbb{T}}[\beta'[\beta]]) \\
\mathtt{check}_{\mathbb{V}} & := & \Lambda\delta :: *.\mathtt{typecase}\ \delta\ \ \langle\Lambda\gamma :: *.\mathtt{check}_{\mathbb{T}}[\gamma]\rangle\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \Lambda\alpha :: *.\Lambda\gamma :: *.\mathtt{marshall}_{\mathbb{V}}[\alpha] \\
& & \quad \mathtt{Var}: \ \Lambda\alpha :: *.\Lambda\gamma :: *.\lambda f : \mathtt{check}_{\mathbb{T}}[\alpha].\lambda v : \mathtt{Var}(\mathtt{String}).v \\
& & \quad \mathtt{Num}: \ \Lambda\alpha :: *.\lambda f : \mathtt{check}_{\mathbb{T}}[\alpha].\lambda n : \mathtt{Num}(\mathtt{int}).n \\
& & \quad \mathtt{Add}: \ \Lambda\alpha :: *.\lambda f_0 : \mathtt{check}_{\mathbb{T}}[\alpha]. \\
& & \qquad \Lambda\beta :: *.\lambda f_1 : \mathtt{check}_{\mathbb{T}}[\beta]. \\
& & \qquad \Lambda\gamma :: *.\lambda a : \mathtt{Add}(\alpha,\beta).\mathtt{Add}(f_0\,[\![a.0]\!]\,, f_1\,[\![a.1]\!]) \\
& & \quad \mathtt{Mul}: \ \Lambda\alpha :: *.\lambda f_0 : \mathtt{check}_{\mathbb{T}}[\alpha]. \\
& & \qquad \Lambda\beta :: *.\lambda f_1 : \mathtt{check}_{\mathbb{T}}[\beta]. \\
& & \qquad \Lambda\gamma :: *.\lambda m : \mathtt{Mul}(\alpha,\beta).\mathtt{Mul}(f_0\,[\![m.0]\!]\,, f_1\,[\![m.1]\!]) \\
\mathtt{x} & := & \mathtt{Var}(\text{"}x\text{"}) \\
\mathtt{y} & := & \mathtt{Var}(\text{"}y\text{"}) \\
\mathtt{JustPair}_{\mathbb{T}} & := & \Lambda\delta :: *.\mathtt{Typecase}\ \delta\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \Lambda\alpha :: *.\mathtt{void} \\
& & \quad \times: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K.\times(\alpha,\beta) \\
\pi_1 & := & \Lambda\delta :: *.\mathtt{Typecase}\ \delta\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \Lambda\alpha :: *.\mathtt{void} \\
& & \quad \times: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K.\alpha \\
\pi_2 & := & \Lambda\delta :: *.\mathtt{Typecase}\ \delta\ \mathtt{of} \\
& & \quad \mathtt{default}: \ \Lambda\alpha :: *.\mathtt{void} \\
& & \quad \times: \ \Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K.\beta \\
\mathtt{add} & := & \Lambda\alpha :: *.\lambda x : \mathtt{JustPair}[\alpha].\mathtt{check}_{\mathbb{V}}[\mathtt{Add}(\pi_1[\alpha],\pi_2[\alpha])]\ \ [\![\mathtt{Add}(x.0, x.1)]\!] \\
\mathtt{mul} & := & \Lambda\alpha :: *.\lambda x : \mathtt{JustPair}[\alpha].\mathtt{check}_{\mathbb{V}}[\mathtt{Mul}(\pi_1[\alpha],\pi_2[\alpha])]\ \ [\![\mathtt{Mul}(x.0, x.1)]\!]
\end{array}
$$

Abbildung A.1: Leicht modifizierter Parser für $\mathbf{V}$ in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

$$
\begin{aligned}
\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!] &= \cdots \\
&= \mathcal{L}[\![(\lambda f : \forall\gamma :: *.\mathtt{void} \to \mathtt{FALSE} \\
&\qquad \Lambda\gamma' :: *.\lambda v : \mathtt{Var}(\mathtt{String}).v) \\
&\qquad (\,\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{String}]]\!]\,)]\!] \\
&= \mathcal{L}[\![(\lambda f : \forall\gamma :: *.\mathtt{void} \to \mathtt{FALSE} \\
&\qquad \Lambda\gamma' :: *.\lambda v : \mathtt{Var}(\mathtt{String}).v) \\
&\qquad (\Lambda\gamma'' :: *.\lambda x : \mathtt{void}.\mathtt{FALSE})]\!] \\
&= \Lambda\gamma :: *.\lambda v : \mathtt{Var}(\mathtt{String}).v
\end{aligned}
$$

Wir leiten zunächst $\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Mul}(\mathtt{Var}(\mathtt{String}), \mathtt{Var}(\mathtt{String}))]]\!]$ her:

$$
\begin{aligned}
&\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Mul}(\mathtt{Var}(\mathtt{String}), \mathtt{Var}(\mathtt{String}))]]\!] \\
&\quad= \mathcal{T}[\![\Lambda\delta :: *.\mathtt{typecase}\ \delta\ \langle\Lambda\gamma :: *.\mathtt{check}_{\mathbb{T}}[\gamma]\rangle\ \mathtt{of}\ \cdots[\mathtt{Mul}(\mathtt{Var}(\mathtt{String}), \mathtt{Var}(\mathtt{String}))]]\!] \\
&\quad= \mathcal{T}[\![\mathtt{typecase}\ \mathtt{Mul}(\mathtt{Var}(\mathtt{String}), \mathtt{Var}(\mathtt{String}))\langle\Lambda\gamma :: *.\mathtt{check}_{\mathbb{T}}[\gamma]\rangle\ \mathtt{of}\ \cdots]\!] \\
&\quad= \mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![\,\mathcal{T}[\![\ \Lambda\alpha :: *.\lambda f_0 : \mathtt{check}_{\mathbb{T}}[\alpha]. \\
&\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \mathtt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\qquad \Lambda\gamma :: *.\lambda m : \mathtt{Mul}(\alpha, \beta).\mathtt{Mul}(f_0\,[\![m.0]\!]\,, f_1\,[\![m.1]\!]) \\
&\qquad\qquad [\mathtt{Var}(\mathtt{String})]\ ]\!] \\
&\qquad\quad (\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!])\ ]\!] \\
&\qquad [\mathtt{Var}(\mathtt{String})]\ ]\!] \\
&\quad (\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!])\ ]\!] \\
&\quad= \mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![\,\mathcal{T}[\![\ \lambda f_0 : \mathtt{check}_{\mathbb{T}}[\mathtt{Var}(\mathtt{String})]. \\
&\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \mathtt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\qquad \Lambda\gamma :: *.\lambda m : \mathtt{Mul}(\mathtt{Var}(\mathtt{String}), \beta).\mathtt{Mul}(f_0\,[\![m.0]\!]\,, f_1\,[\![m.1]\!])\ ]\!] \\
&\qquad\quad (\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!])\ ]\!] \\
&\qquad [\mathtt{Var}(\mathtt{String})]\ ]\!] \\
&\quad (\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!])\ ]\!] \\
&\quad= \mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![\lambda f_0 :\ \mathcal{T}[\![\mathtt{check}_{\mathbb{T}}[\mathtt{Var}(\mathtt{String})]]\!]. \\
&\qquad\quad \Lambda\beta :: *.\lambda f_1 : \mathtt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\quad \Lambda\gamma :: *.\lambda m : \mathtt{Mul}(\mathtt{Var}(\mathtt{String}), \beta).\mathtt{Mul}(f_0\,[\![m.0]\!]\,, f_1\,[\![m.1]\!]) \\
&\qquad\quad (\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!])\ ]\!] \\
&\qquad [\mathtt{Var}(\mathtt{String})]\ ]\!] \\
&\quad (\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!])\ ]\!] \\
&\quad= \mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![\lambda f_0 : \mathtt{Var}(\mathtt{String}) \to \mathtt{Var}(\mathtt{String}) \\
&\qquad\quad \Lambda\beta :: *.\lambda f_1 : \mathtt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\quad \Lambda\gamma :: *.\lambda m : \mathtt{Mul}(\mathtt{Var}(\mathtt{String}), \beta).\mathtt{Mul}(f_0\,[\![m.0]\!]\,, f_1\,[\![m.1]\!]) \\
&\qquad\quad (\,\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!]\,)\ ]\!] \\
&\qquad [\mathtt{Var}(\mathtt{String})]\ ]\!] \\
&\quad (\,\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Var}(\mathtt{String})]]\!]\,)\ ]\!] \\
&\quad= \cdots
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String}))]]\!] \\
&\quad = \cdots \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\ \mathcal{L}[\![\ \lambda f_0 : \texttt{Var}(\texttt{String}) \to \texttt{Var}(\texttt{String}) \\
&\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \texttt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\qquad \Lambda\gamma :: *.\lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta).\texttt{Mul}(f_0\,[\![m.0]\!]\,, f_1\,[\![m.1]\!]) \\
&\qquad\qquad (\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\ ]\!] \\
&\qquad\quad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad (\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\ ]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\ \mathcal{T}[\![\ \Lambda\beta :: *.\lambda f_1 : \texttt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\qquad \Lambda\gamma :: *.\lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta). \\
&\qquad\qquad \texttt{Mul}((\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\,[\![m.0]\!]\,, \\
&\qquad\qquad\quad f_1\,[\![m.1]\!])\ ]\!] \\
&\qquad\quad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad (\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\ ]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\Lambda\beta :: *.\lambda f_1 : \texttt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\quad \Lambda\gamma :: *.\lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta). \\
&\qquad\quad \texttt{Mul}(\ \mathcal{T}[\![(\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\,[\![m.0]\!]\ ]\!]\,, \\
&\qquad\qquad f_1\,[\![m.1]\!]) \\
&\qquad\quad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad (\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\ ]\!]
\end{aligned}
$$

An dieser Stelle kann man den Effekt unserer Änderung am Parser sehen: Wir nutzen Überladungsapplikation aus, um den partiellen Auswerter zu einem β-Reduktionsschritt zu zwingen. Ohne Überladungsapplikation und Dummy-Typargument würden wir anstelle des Redex $\mathcal{T}[\![(\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\,[\![m.0]\!]]\!]$ den Term $(\lambda v : \texttt{Var}(\texttt{String}))(m.0)$ vorfinden und die Anwendung des String-Parsers auf das Argument $m.0$ müßte zur Laufzeit erfolgen.

Wir fahren mit der Reduktion fort:

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String}))]]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\Lambda\beta :: *.\lambda f_1 : \texttt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\quad \Lambda\gamma :: *.\lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta). \\
&\qquad\quad \texttt{Mul}(\ \mathcal{T}[\![(\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\,[\![m.0]\!]\ ]\!]\,, \\
&\qquad\qquad f_1\,[\![m.1]\!]) \\
&\qquad\quad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad (\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\ ]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\Lambda\beta :: *.\lambda f_1 : \texttt{check}_{\mathbb{T}}[\beta]. \\
&\qquad\quad \Lambda\gamma :: *.\lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta). \\
&\qquad\quad \texttt{Mul}(\mathcal{L}[\![\ \mathcal{T}]\!](\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)[\texttt{Var}(\texttt{String})]\ ]\!]\,(\ \mathcal{T}[\![m.0]\!]\ )\ ]\!] \\
&\qquad\qquad f_1\,[\![m.1]\!]) \\
&\qquad\quad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad (\Lambda\gamma :: *.\lambda v : \texttt{Var}(\texttt{String}).v)\ ]\!] \\
&\quad = \cdots
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\mathtt{Var}(\mathtt{String}))]]\!]\\
&\quad = \cdots\\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\Lambda\beta :: *.\lambda f_1 : \mathtt{check}_{\mathbb{T}}[\beta].\\
&\qquad\qquad \Lambda\gamma :: *.\lambda m : \mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\beta).\\
&\qquad\qquad \mathtt{Mul}(\ \mathcal{L}[\![(\lambda v : \mathtt{Var}(\mathtt{String}).v)\ \ m.0\ ]\!]\\
&\qquad\qquad\quad f_1\,[\![m.1]\!])\\
&\qquad\qquad [\mathtt{Var}(\mathtt{String})]\ ]\!]\\
&\qquad\quad (\Lambda\gamma :: *.\lambda v : \mathtt{Var}(\mathtt{String}).v)\ ]\!]\\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\Lambda\beta :: *.\lambda f_1 : \mathtt{check}_{\mathbb{T}}[\beta].\\
&\qquad\qquad \Lambda\gamma :: *.\lambda m : \mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\beta).\\
&\qquad\qquad \mathtt{Mul}(m.0,\\
&\qquad\qquad\quad f_1\,[\![m.1]\!])\\
&\qquad\qquad [\mathtt{Var}(\mathtt{String})]\ ]\!]\\
&\qquad\quad (\Lambda\gamma :: *.\lambda v : \mathtt{Var}(\mathtt{String}).v)\ ]\!]
\end{aligned}
$$

Die Reduktion der noch anstehenden Typ- und Wertapplikation verläuft analog zur ersten, so daß wir als Ergebnis

$$
\begin{aligned}
&\mathcal{T}[\![\mathtt{check}_{\mathbb{V}}[\mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\mathtt{Var}(\mathtt{String}))]]\!]\\
&\qquad = \Lambda\gamma :: *.\lambda m : \mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\mathtt{Var}(\mathtt{String})).\mathtt{Mul}(m.0, m.1)
\end{aligned}
$$

erhalten.

Wir verifizieren, daß $\mathtt{check}_{\mathbb{T}}[\mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\mathtt{Var}(\mathtt{String}))]$ einen zu dieser Funktion passenden Typ berechnet, dabei nutzen wir aus, daß offenbar

$$
\begin{aligned}
\mathtt{Res}[\mathtt{check}_{\mathbb{T}}[\mathtt{Var}(\mathtt{String})][\mathtt{int}]\ ] &= \mathtt{Res}[(\forall\gamma :: *.\mathtt{Var}(\mathtt{String}) \to \mathtt{Var}(\mathtt{String}))[\mathtt{int}]]\\
&= \mathtt{Var}(\mathtt{String})
\end{aligned}
$$

gilt:

$$
\begin{aligned}
&\mathcal{T}[\![\mathtt{check}_{\mathbb{T}}[\mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\mathtt{Var}(\mathtt{String}))]]\!]\\
&\quad = (\Lambda\alpha :: *.\Lambda\alpha' :: K.\Lambda\beta :: *.\Lambda\beta' :: K.\\
&\qquad \forall\gamma :: *.\mathtt{Mul}(\alpha,\beta) \to \mathtt{Mul}(\mathtt{Res}(\alpha'[\mathtt{int}]),\mathtt{Res}(\beta'[\mathtt{int}])\\
&\qquad)[\mathtt{Var}(\mathtt{String})][\mathtt{check}_{\mathbb{T}}[\mathtt{Var}(\mathtt{String})]][\mathtt{Var}(\mathtt{String})][\mathtt{check}_{\mathbb{T}}[\mathtt{Var}(\mathtt{String})]]\\
&\quad = \forall\gamma :: *.\mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\mathtt{Var}(\mathtt{String})) \to \mathtt{Mul}(\mathtt{Var}(\mathtt{String}),\mathtt{Var}(\mathtt{String}))
\end{aligned}
$$

Mit diesen Ergebnissen und

$$
\begin{aligned}
\pi_1[\mathtt{Var}(\mathtt{String}) \times \mathtt{Var}(\mathtt{String})] &= \mathtt{Var}(\mathtt{String})\\
\pi_2[\mathtt{Var}(\mathtt{String}) \times \mathtt{Var}(\mathtt{String})] &= \mathtt{Var}(\mathtt{String})
\end{aligned}
$$

fällt die Reduktion von $\mathcal{T}[\![\mathtt{mul}\,[\![(x,y)]\!]\ ]\!]$ leicht:

$$
\begin{aligned}
\mathcal{T}[\![\texttt{mul}\,[\![(x,y)]\!]]\!] &= \mathcal{L}[\![\,\mathcal{T}[\![\,(\Lambda\alpha :: *.\lambda x : \texttt{JustPair}[\alpha].\\
&\qquad \texttt{check}_{\mathbb{V}}[\texttt{Mul}(\pi_1[\alpha]\times \pi_2[\alpha])\ ][\![\texttt{Mul}(x.0,x.1)]\!])\\
&\qquad [\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})]\ ]\!]\\
&\qquad (x,y)\ ]\!]\\
&= \mathcal{L}[\![\,\mathcal{T}[\![\,\lambda x : \texttt{JustPair}[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})].\\
&\qquad \texttt{check}_{\mathbb{V}}[\texttt{Mul}(\pi_1[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})]\times \pi_2[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})])\ ][\![\texttt{Mul}(x.0,x.1)]\!]\ ]\!]\\
&\qquad (x,y)\ ]\!]\\
&= \mathcal{L}[\![\lambda x : \mathcal{T}[\![\texttt{JustPair}[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})]]\!].\\
&\quad \mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{Mul}(\pi_1[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})]\times \pi_2[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})])][\![\texttt{Mul}(x.0,x.1)]\!]\ ]\!]\\
&\quad (x,y)\ ]\!]\\
&= \mathcal{L}[\![\lambda x : \texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})].\\
&\quad \mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{Mul}(\pi_1[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})]\ , \pi_2[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})])][\![\texttt{Mul}(x.0,x.1)]\!]\ ]\!]\\
&\quad (x,y)\ ]\!]\\
&= \mathcal{L}[\![\lambda x : \texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})].\\
&\quad \mathcal{L}[\![\,\mathcal{T}[\![\texttt{check}_{\mathbb{V}}[\texttt{Mul}(\pi_1[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})]\ , \pi_2[\texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})])][\texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String}))]\ ]\!]\\
&\qquad \mathcal{T}[\![\texttt{Mul}(x.0,x.1)]\!]\ ]\!]\\
&\quad (x,y)\ ]\!]\\
&= \mathcal{L}[\![\lambda x : \texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})].\\
&\quad \mathcal{L}[\![\,\mathcal{T}[\![\Lambda\gamma :: *.\lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String})).\texttt{Mul}(m.0,m.1)\\
&\qquad [\texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String}))]\ ]\!]\\
&\qquad \texttt{Mul}(x.0,x.1)\ ]\!]\\
&\quad (x,y)\ ]\!]\\
&= \mathcal{L}[\![\lambda x : \texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})].\\
&\quad \mathcal{L}[\![\,\lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String})).\texttt{Mul}(m.0,m.1)\\
&\qquad \texttt{Mul}(x.0,x.1)\ ]\!]\\
&\quad (x,y)\ ]\!]\\
&= \mathcal{L}[\![\lambda x : \texttt{Var}(\texttt{String})\times\texttt{Var}(\texttt{String})].\\
&\quad \mathcal{L}[\![\,\lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String})).\texttt{Mul}(m.0,m.1)\\
&\qquad \texttt{Mul}(x.0,x.1)\ ]\!]\\
&\quad (x,y)\ ]\!]\\
&= \cdots
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}[\![\mathtt{mul}\,[\!(x,y)]\!]] &= \cdots \\
&= \mathcal{L}[\![\lambda x : \mathtt{Var(String)} \times \mathtt{Var(String)}]. \\
&\quad\ \mathcal{T}[\![\mathtt{Mul}(\mathtt{Mul}(x.0, x.1).0, \mathtt{Mul}(x.0, x.1).1)]\!] \\
&\quad\ (x, y)\ ]\!] \\
&= \mathcal{L}[\![\lambda x : \mathtt{Var(String)} \times \mathtt{Var(String)}]. \\
&\quad\ \mathtt{Mul}(\ \mathcal{T}[\![\mathtt{Mul}(x.0, x.1).0]\!]\ ,\ \mathcal{T}[\![\mathtt{Mul}(x.0, x.1).1]\!]\ ) \\
&\quad\ (x, y)\ ]\!] \\
&= \mathcal{L}[\![\ \lambda x : \mathtt{Var(String)} \times \mathtt{Var(String)}]. \\
&\quad\ \mathtt{Mul}(x.0, x.1) \\
&\quad\ (x, y)\ ]\!] \\
&= \mathcal{T}[\![\mathtt{Mul}((x, y).0, (x, y).1)]\!] \\
&= \mathtt{Mul}(\ \mathcal{T}[\![(x, y).0]\!]\ ,\ \mathcal{T}[\![(x, y).1]\!]\ ) \\
&= \mathtt{Mul}(x, y)
\end{aligned}
$$

Die Herleitung zu $\mathcal{T}[\![\mathtt{add}\,[\!(x, \mathtt{mul}\,[\!(x, y)]\!]\ )]\!]$ verläuft analog. Im Ergebnis erhält man

$$
\begin{aligned}
&\mathcal{T}[\![\mathtt{add}\,[\!(x, \mathtt{mul}\,[\!(x, y)]\!]\ )]\!] \\
&\quad = \mathtt{Add}(x, \mathtt{Mul}(x, y))
\end{aligned}
$$

Das Parsing (bzw. die Analyse eines **V**-Terms auf Korrektheit) wird komplett vom partiellen Auswerter bewerkstelligt.

A.3 Partielle Auswertung von $\mathtt{Add}(x, \mathtt{Mul}(x, y))$ in System $\mathbf{F}^{\mathbf{SA}}_{\omega,1}$

Abbildung A.2 zeigt einen im Vergleich zu Abbildung 4.3 auf Seite 86 leicht modifizierten Interpreter für **V**-Terme. Wir haben zwei Änderungen vorgenommen:

- Der Codegenerator übernimmt neben dem Typ des zu berechnenden Terms eine Umgebung. Die generierten Interpreter können dann auf diese zurückgreifen, so daß man sich zahlreiche β-Reduktionsschritte sparen kann.
- Wie schon beim modifizierten Parser, erwarten die generierten Interpreter als erstes Argument einen Typ – sind also polymorph. Generierte Funktionen können dadurch via Überladungsapplikation aufgerufen werden, für die der partielle Auswerter einen β-Reduktionsschritt vollzieht.

Die Funktion `interV` dient in erster Linie dazu, dem Anwender die gewohnte Schnittstelle zum Interpreter anbieten zu können.

Wir berechnen zunächst die Interpreterfunktion für Variablen.

$$
\begin{array}{rcl}
\texttt{vInterGen} & := & \lambda e : \texttt{String} \to \texttt{int}. \\
& & \Lambda\gamma :: *.\texttt{typecase}\ \gamma\ \ \langle \Lambda\delta :: *.\delta \to (\texttt{String} \to \texttt{int}) \to \texttt{int}\rangle\ \texttt{of} \\
& & \qquad \texttt{default}: \ \ \Lambda\alpha :: *.\Lambda\eta :: *.\lambda x : \alpha.0 \\
& & \qquad\quad \texttt{Num}: \ \ \Lambda\alpha :: *.\lambda f : \forall\delta :: *.\alpha \to\!\!\to \texttt{int}. \\
& & \qquad\qquad\qquad \Lambda\eta :: *.\lambda n : \texttt{Num}(\alpha). \\
& & \qquad\qquad\qquad n.0 \\
& & \qquad\quad \texttt{Var}: \ \ \Lambda\alpha :: *.\lambda f : \forall\delta :: *.\alpha \to \texttt{int}. \\
& & \qquad\qquad\qquad \Lambda\eta :: *.\lambda v : \texttt{Var}(\texttt{String}). \\
& & \qquad\qquad\qquad e(v.0) \\
& & \qquad\quad \texttt{Add}: \ \ \Lambda\alpha :: *.\lambda f_0 : \forall\delta :: *.\alpha \to \texttt{int}. \\
& & \qquad\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int} \\
& & \qquad\qquad\qquad \Lambda\eta :: *.\lambda a : \texttt{Add}(\alpha,\beta). \\
& & \qquad\qquad\qquad (f_0\,[\![a.0]\!]) + (f_1\,[\![a.1]\!]) \\
& & \qquad\quad \texttt{Mul}: \ \ \Lambda\alpha :: *.\lambda f_0 : \forall\delta :: *.\alpha \to \texttt{int}. \\
& & \qquad\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int} \\
& & \qquad\qquad\qquad \Lambda\eta :: *.\lambda a : \texttt{Mul}(\alpha,\beta). \\
& & \qquad\qquad\qquad (f_0\,[\![a.0]\!]) * (f_1\,[\![a.1]\!]) \\
\texttt{vInter} & := & \Lambda\delta :: *.\lambda t : \delta.\lambda e : \texttt{String} \to \texttt{int}.\texttt{vInterGen}\ e\ [\delta]\ [\delta]\ t
\end{array}
$$

Abbildung A.2: Ein leicht modifizierter Interpreter für $\mathbf{V}$ in System $\mathbf{F}^{\mathrm{SA}}_{\omega,1}$.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathcal{T}[\![(\ \Lambda\delta :: *.\texttt{typecase}\ \gamma\langle\Lambda\gamma :: *.\gamma \to \texttt{int}\ \texttt{of}\ \cdots)} \\
\multicolumn{3}{l}{\qquad [\texttt{Var}(\texttt{String})]\ \ ]\!]} \\
& = & \mathcal{T}[\![\texttt{typecase}\ \ \texttt{Var}(\texttt{String}) \\
& & \qquad\qquad\qquad \langle\Lambda\gamma :: *.\gamma \to \texttt{int} \\
& & \qquad\qquad\qquad\ \texttt{of}\ \cdots]\!] \\
& = & \mathcal{L}[\![\ \mathcal{T}[\![\ (\Lambda\alpha : *.\lambda f : \alpha \to \texttt{int}. \\
& & \qquad\qquad \Lambda\delta :: *.\lambda v : \texttt{Var}(\texttt{String}).e(v.0)\ \ ) \\
& & \qquad\qquad [\texttt{String}]\ \ ]\!] \\
& & \qquad (\mathcal{T}[\![\texttt{typecase String of}\ \cdots]\!])\ \ ]\!] \\
& = & \mathcal{L}[\![(\lambda f : \texttt{String} \to \texttt{int}. \\
& & \qquad \Lambda\delta :: *.\lambda v : \texttt{Var}(\texttt{String}).e(v.0)\ \ ) \\
& & \qquad (\ \mathcal{T}[\![\texttt{typecase String of}\ \cdots]\!]\ )\ \ ]\!] \\
& = & \mathcal{L}[\![\ (\lambda f : \texttt{String} \to \texttt{int}. \\
& & \qquad\quad \Lambda\delta :: *.\lambda n : \texttt{Var}(\texttt{String}).n.0) \\
& & \qquad\quad (\lambda x : \texttt{String}.\lambda e : \texttt{String} \to \texttt{int}.0)\ \ ]\!] \\
& = & \Lambda\delta :: *.\lambda v : \texttt{Var}(\texttt{String}).e(v.0)
\end{array}
$$

Die Interpreterfunktion für den Teilterm $\texttt{Mul}(x, y)$ leiten wir ausführlich her. Die Berechnung der Interpreterfunktion zu $\texttt{Add}(x, \texttt{Mul}(x, y))$ verläuft nach genau demselben Schma.

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{vInter}[\texttt{Mul}(\texttt{Var}(\texttt{String}), \texttt{Var}(\texttt{String}))]\ \texttt{Mul}(x.y)]\!] \\
&\quad = \mathcal{T}[\![(\ \Lambda\delta :: *.\texttt{typecase}\ \gamma\langle\Lambda\gamma :: *.\gamma \to \texttt{int of}\ \cdots) \\
&\qquad [\texttt{Mul}(\texttt{Var}(\texttt{String}), \texttt{Var}(\texttt{String}))]]\!] \\
&\quad = \mathcal{T}[\![\texttt{typecase}\ \ \texttt{Mul}(\texttt{Var}(\texttt{String}), \texttt{Var}(\texttt{String})) \\
&\qquad\qquad \langle\Lambda\gamma :: *.\gamma \to \texttt{int} \\
&\qquad\qquad \texttt{of}\ \cdots]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![\ \mathcal{T}[\![\ \Lambda\alpha :: *.\lambda f_0 : \forall\delta :: *.\alpha \to \texttt{int}. \\
&\qquad\qquad\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int}. \\
&\qquad\qquad\qquad\qquad \lambda m : \texttt{Mul}(\alpha, \beta). \\
&\qquad\qquad\qquad\qquad (f_0\ [\![(m.0)]\!]\ )* \\
&\qquad\qquad\qquad\qquad (f_1\ [\![(m.1)]\!]\ ) \\
&\qquad\qquad\qquad\qquad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\qquad\qquad (\mathcal{T}[\![\texttt{typecase}\ \texttt{Var}(\texttt{String})\ \texttt{of}\ \ldots]\!])\ ]\!] \\
&\qquad\qquad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\qquad (\mathcal{T}[\![\texttt{typecase}\ \texttt{Var}(\texttt{String})\ \texttt{of}\ \ldots]\!])\ ]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\mathcal{L}[\![\lambda f_0 : \forall\delta :: *.\texttt{Var}(\texttt{String}) \to \texttt{int}. \\
&\qquad\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int}. \\
&\qquad\qquad\qquad \lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}), \beta). \\
&\qquad\qquad\qquad (f_0\ [\![(m.0)]\!]\ )* \\
&\qquad\qquad\qquad (f_1\ [\![(m.1)]\!]\ ) \\
&\qquad\qquad\qquad (\ \mathcal{T}[\![\texttt{typecase}\ \texttt{Var}(\texttt{String})\ \texttt{of}\ \ldots]\!]\ )\ ]\!] \\
&\qquad\qquad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\qquad (\mathcal{T}[\![\texttt{typecase}\ \texttt{Var}(\texttt{String})\ \texttt{of}\ \ldots]\!])\ ]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\ \mathcal{L}[\![\ \lambda f_0 : \forall\delta :: *.\texttt{Var}(\texttt{String}) \to \texttt{int}. \\
&\qquad\qquad\qquad\ \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int}. \\
&\qquad\qquad\qquad\ \lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}), \beta). \\
&\qquad\qquad\qquad\ (f_0\ [\![(m.0)]\!]\ )* \\
&\qquad\qquad\qquad\ (f_1\ [\![(m.1)]\!]\ ) \\
&\qquad\qquad\qquad\ (\Lambda\delta :: *.\lambda v : \texttt{Var}(\texttt{String}).e(v.0)\ \ )\ ]\!] \\
&\qquad\qquad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\qquad (\mathcal{T}[\![\texttt{typecase}\ \texttt{Var}(\texttt{String})\ \texttt{of}\ \ldots]\!])\ ]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\ \mathcal{T}[\![\ \lambda f_0 : \forall\delta :: *.\texttt{Var}(\texttt{String}) \to \texttt{int}. \\
&\qquad\qquad\qquad\ \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int}. \\
&\qquad\qquad\qquad\ \lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}), \beta). \\
&\qquad\qquad\qquad\ (\Lambda\delta :: *.\lambda v : \texttt{Var}(\texttt{String}).e(v.0))\ [\![(m.0)]\!]\ )* \\
&\qquad\qquad\qquad\ (f_1\ [\![(m.1)]\!]\ ) \\
&\qquad\qquad\qquad\ ]\!] \\
&\qquad\qquad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\qquad (\mathcal{T}[\![\texttt{typecase}\ \texttt{Var}(\texttt{String})\ \texttt{of}\ \ldots]\!])\ ]\!] \\
&\quad = \cdots
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{vInter}[\texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String}))]\ \texttt{Mul}(x.y)]\!] \\
&\quad = \cdots \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\lambda f_0 : \forall\delta :: *.\texttt{Var}(\texttt{String}) \to \texttt{int}. \\
&\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int}. \\
&\qquad\qquad \lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta). \\
&\qquad\qquad \mathcal{T}[\![(\Lambda\delta :: *.\lambda v : \texttt{Var}(\texttt{String}).e(v.0))\,[\![(m.0)]\!]\ ]\!]\ )* \\
&\qquad\qquad (f_1\,[\![(m.1)]\!]\ ) \\
&\qquad\qquad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\quad (\mathcal{T}[\![\texttt{typecase Var(String) of } \ldots]\!])\ ]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\lambda f_0 : \forall\delta :: *.\texttt{Var}(\texttt{String}) \to \texttt{int}. \\
&\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int}. \\
&\qquad\qquad \lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta). \\
&\qquad\qquad \mathcal{L}[\!\,\mathcal{T}[\![(\Lambda\delta :: *.\lambda v : \texttt{Var}(\texttt{String}).e(v.0))[\texttt{Var}(\texttt{String})]\,(\mathcal{T}[\![m.0]\!])\ )* \\
&\qquad\qquad (f_1\,[\![(m.1)]\!]\ ) \\
&\qquad\qquad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\quad (\mathcal{T}[\![\texttt{typecase Var(String) of } \ldots]\!])\ ]\!] \\
&\quad = \mathcal{L}[\![\mathcal{T}[\![\lambda f_0 : \forall\delta :: *.\texttt{Var}(\texttt{String}) \to \texttt{int}. \\
&\qquad\qquad \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int}. \\
&\qquad\qquad \lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta). \\
&\qquad\qquad \mathcal{L}[\![\lambda v : \texttt{Var}(\texttt{String}).e(v.0))(\mathcal{T}[\![m.0]\!])\ )* \\
&\qquad\qquad (f_1\,[\![(m.1)]\!]\ ) \\
&\qquad\qquad [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\quad (\mathcal{T}[\![\texttt{typecase Var(String) of } \ldots]\!])\ ]\!] \\
&\quad = \mathcal{L}[\![\,\mathcal{T}[\![\ \lambda f_0 : \forall\delta :: *.\texttt{Var}(\texttt{String}) \to \texttt{int}. \\
&\qquad\qquad\ \Lambda\beta :: *.\lambda f_1 : \forall\delta :: *.\beta \to \texttt{int}. \\
&\qquad\qquad\ \lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\beta). \\
&\qquad\qquad\ e(m.0.0) * \\
&\qquad\qquad\ (f_1\,[\![(m.1)]\!]\ ) \\
&\qquad\qquad\ [\texttt{Var}(\texttt{String})]\ ]\!] \\
&\qquad\quad (\mathcal{T}[\![\texttt{typecase Var(String) of } \ldots]\!])\ ]\!] \\
&\quad = \cdots \\
&\quad = \lambda m : \texttt{Mul}(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String})).e(m.0.0) * e(m.0.1)
\end{aligned}
$$

Wir haben die Herleitung abgekürzt, da die Auswertung der noch anstehenden Typ- und Wertapplikation völlig analog zur ersten läuft.

Die partielle Auswertung von $\texttt{Add}(x,\texttt{Mul}(x,y))$ verläuft nach demselben Muster und liefert die Funktion

$$
\begin{aligned}
&\lambda m : \texttt{Add}(\texttt{Var}(\texttt{String})(\texttt{Var}(\texttt{String}),\texttt{Var}(\texttt{String})). \\
&e(m.1) + (e(m.0.0) * e(m.1.1)))
\end{aligned}
$$

als Residuum.

A.4 Demonstration des abstrakten CoreEML-Interpreters

In diesem Abschnitt wollen wir die Arbeitsweise des abstrakten CoreEML-Interpreters an einem Beispiel demonstrieren.
Gegeben sei das folgende EML-Programm:

```
( fac(x) = If(x==0,1,x*fac(x-1) ) )[ fac(10) ]
```

Der Interpreter beginnt seine Arbeit im Zustand

$$\{|\texttt{App}(\texttt{fac}, \texttt{int})|\} \quad \emptyset$$

Ein Applikationsknoten liegt auf dem Stack. Gemäß der ersten Prämisse der Regel **App** wird ein neuer Interpreter gestartet, dessen Stack lediglich den Typ `int` enthält. Mit der Regel **Const** kann man schließen, daß `int` eine Normalform ist und so wird mit der Bearbeitung der zweiten Prämisse fortgefahren. Ein neuer Interpreter wird im Zustand

$$\{|\texttt{fac} : \texttt{int}|\} \quad \emptyset$$

gestartet.
Das EML-Skript enthält eine Superkombinatordefinition zur funktionalen Variable `fac` und es liegen ausreichend Argumente auf dem Stack, um die rechte Seite von `fac` zu instanziieren. Da $\{|\texttt{fac} : \texttt{int}|\} \notin \emptyset$ gilt, kann die Regel **Sc** Anwendung finden. Wiederum wird ein neuer Interpreter gestartet. Dessen Stack enthält als einziges Element die mit `int` instanziierte rechte Seite der Definition von `fac`. Die Aufrufspur wird um den Eintrag $\{|\texttt{fac} : \texttt{int}|\} \notin \emptyset$ erweitert:

$$\left\{\left|\begin{array}{l}\texttt{App}(\texttt{App}(\texttt{App}(\texttt{OpIf},\\ \qquad\texttt{App}(\texttt{App}(\texttt{OpEq}, \texttt{int}), \texttt{int})),\\ \quad\texttt{int}),\\ \texttt{App}(\texttt{App}(\texttt{OpMult}, \texttt{int}),\\ \quad\texttt{App}(\texttt{fac}, \texttt{App}(\texttt{App}(\texttt{OpSub}, \texttt{int}), \texttt{int}))))\end{array}\right|\right\} \quad \{\{|\texttt{fac} : \texttt{int}|\}\}$$

Das Argument des äußeren Applikationsknotens liegt noch nicht in ausgewerteter Form vor und wird gemäß der ersten Prämisse der Regel **App** reduziert. Der Interpreter befindet sich im Zustand

$$\left\{\left|\begin{array}{l}\texttt{App}(\texttt{App}(\texttt{OpMult}, \texttt{int}),\\ \quad\texttt{App}(\texttt{fac}, \texttt{App}(\texttt{App}(\texttt{OpSub}, \texttt{int}), \texttt{int})))\end{array}\right|\right\} \quad \{\{|\texttt{fac} : \texttt{int}|\}\}$$

Abermals ist ein Funktionsargument zu reduzieren:

$$\left\{\left|\begin{array}{l}\texttt{App}(\texttt{fac},\\ \quad\texttt{App}(\texttt{App}(\texttt{OpSub}, \texttt{int}), \texttt{int}))\end{array}\right|\right\} \quad \{\{|\texttt{fac} : \texttt{int}|\}\}$$

Durch wiederholtes Anwenden der Regel **App** gelangt der Interpreter in den Zustand

$$\{\!|\texttt{OpSub} : \texttt{int} : \texttt{int}|\!\}\ \ \{\{\!|\texttt{fac} : \texttt{int}|\!\}\}$$

Auf der Stackspitze liegt das Funktionssymbol der eingebauten Subtraktionsfunktion. Wegen $bop_{\mathbb{T}}[\texttt{OpSub} \times \texttt{int} \times \texttt{int}] = \texttt{int}$ (siehe Abbildung 9.1) kann der Interpreter mit der Regel $\mathbf{BinaryOp}_{\texttt{OpSub}}$ in den Zustand

$$\{\!|\texttt{int}|\!\}\ \ \{\{\!|\texttt{fac} : \texttt{int}|\!\}\}$$

übergehen. Das Argument, das der Fakultätsfunktion im rekursiven Aufruf übergeben wird steht damit fest und es kann mit der Berechnung der Normalform zu $\texttt{fac}[\texttt{int}]$ fortgefahren werden. Der Interpreter wird dazu in den Zustand

$$\{\!|\texttt{fac} : \texttt{int}|\!\}\ \ \{\{\!|\texttt{fac} : \texttt{int}|\!\}\}$$

versetzt. `fac` ist ein Superkombinatorsymbol und es liegen ausreichend Argumente auf dem Stack. Die Regel **Sc** kommt diesmal jedoch nicht in Frage, da die *call trace* ein Element $\{\!|\texttt{fac} : \texttt{int}|\!\}$ enthält. Mit Regel **ScRek** geht der Interpreter daher über in den Zustand

$$\{\!|\bot|\!\}\ \ \{\{\!|\texttt{fac} : \texttt{int}|\!\}\}$$

Kehren wir zurück zum Anfang der Berechnung des else-Zweiges. Der Interpreter war im Zustand

$$\left\{\left|\begin{array}{l}\texttt{App}(\texttt{App}(\texttt{OpMult}, \texttt{int}),\\ \quad\texttt{App}(\texttt{fac}, \texttt{App}(\texttt{App}(\texttt{OpSub}, \texttt{int}), \texttt{int})))\end{array}\ : [] \right|\right\}\ \ \{\{\!|\texttt{fac} : \texttt{int}|\!\}\}$$

und wir mußten zunächst das Argument des Applikationsknotens reduzieren. Das haben wir soeben getan und herausgefunden, daß dies zum Typsymbol $\bot$ führt. Der Interpreter setzt seine Arbeit mit der zweiten Prämisse der Regel **App** fort:

$$\{\!|\texttt{App}(\texttt{OpMult}, \texttt{int}) : \bot : \{\{\!|\texttt{fac} : \texttt{int}|\!\}\}|\!\}$$

Nach abermaligem Anwenden der Regel **App** sieht der Interpreter auf der Stackspitze das Funktionssymbol `OpMult` und die Argumente `int` und $\bot$. Da $bop_{\mathbb{T}}[\texttt{OpMult} \times \texttt{int} \times \bot] = \bot$
Damit ist der else-Zweig von `If` ausgewertet und die Arbeit wird mit der zweiten Prämisse der Regel **App** fortgesetzt:

$$\left\{\left|\begin{array}{l}\texttt{App}(\texttt{App}(\texttt{OpIf},\\ \qquad\texttt{App}(\texttt{App}(\texttt{OpEq}, \texttt{int}), \texttt{int}))\\ \quad\texttt{int})\end{array}\ : \bot \right|\right\}\ \ \{\{\!|\texttt{fac} : \texttt{int}|\!\}\}$$

Die erste Prämisse der Regel **App** findet in `int` eine Normalform zum Argument. Vor der Anwendung der zweiten Prämisse ist der Interpreter im Zustand

$$\left\{\left|\begin{array}{l}\texttt{App}(\texttt{OpIf},\\ \quad\texttt{App}(\texttt{App}(\texttt{OpEq},\texttt{int}),\texttt{int}))\end{array}\right. : \texttt{int} : \bot\right\}\ \ \{\{|\texttt{fac} : \texttt{int}|\}\}$$

und es wird mit der Auswertung des Arguments fortgefahren. Wie man leicht nachvollzieht gilt

$$\{|\texttt{App}(\texttt{App}(\texttt{OpEq},\texttt{int}),\texttt{int}))|\}\ \ \{\{|\texttt{fac} : \texttt{int}|\}\} \Downarrow_{\mathbb{T}} \{|\texttt{bool}|\}\ \ \{\{|\texttt{fac} : \texttt{int}|\}\}$$

und der Interpreter kann zur Abarbeitung der zweiten Prämisse von **App** in den Zustand

$$\{|\texttt{OpIf} : \texttt{bool} : \texttt{int} : \bot|\}\ \ \{\{|\texttt{fac} : \texttt{int}|\}\}$$

übergehen. Es gilt

$$\texttt{If}_{\mathbb{T}}[\texttt{bool} \times \texttt{bool} \times \texttt{int} \times \bot] = \texttt{int}$$

da $\bigsqcup\{\texttt{int}, \bot\} = \texttt{int}$. Somit erhalten wir insgesamt:

$$\{|\texttt{App}(\texttt{fac},\texttt{int})|\}\ \ \emptyset \Downarrow_{\mathbb{T}} \{|\texttt{int}|\}\ \ \emptyset$$

und schließen, daß unser EML-Programm einen `int`-Wert als Ergebnis liefert.

Anhang B

Eingesetzte Metaprogramme

In diesem Kapitel listen wir einige der Metaprogramme auf, die wir nicht im Fließtext wiedergegeben haben.

B.1 Standardprogramme

```
template <bool C,typename T,typename E> struct
IF { typedef T RET;};
template <typename T,typename E> struct
IF<false,T,E> { typedef E RET; };

template <typename T1,typename T2> struct
Equal { enum { Ret=false }; };
template <typename T> struct
Equal<T,T> { enum {Ret=true};};
```

B.2 Listen und Listenoperationen

```
#ifndef TMPL_LIST
#define TMPL_LIST

#ifdef TMPL_LIST_NAMESPACE
namespace TML {
#endif

/*
+============================================================================+
| Konstruktoren N und C zum Aufbau von Listen                                |
+============================================================================+
| Die Struktur der Listen spiegelt sich in ihrem Typ wider. Listen können    |
```

```
/ daher durch intensionelle Typanalyse zur Übersetzungszeit manipuliert werden|
| bzw. kann Programmcode zu ihrer Manipulation generiert werden.              |
+=============================================================================+ */
struct N { };

template <typename HEAD,typename TAIL>
struct C {
  typedef HEAD Head_t;
  typedef TAIL Tail_t;

  C() {}
  C(const HEAD& h,const TAIL& t) : head(h),tail(t)   {}
  C(const C& rhs) : head(rhs.head),tail(rhs.tail) {}

  inline const HEAD& getHead() const { return head; }
  inline const TAIL& getTail() const { return tail; }

inline HEAD& getHead() { return head; }
inline TAIL& getTail() { return tail; }

C<HEAD,TAIL>& operator=(const C<HEAD,TAIL>& rhs) {
  head=rhs.getHead();
tail=rhs.getTail();
return *this;
}

  HEAD head;
  TAIL tail;
};

/*
+=============================================================================+
| cons                                                                        |
+=============================================================================+
| Die folgenden cons-Funktionen dienen dem Aufbau von Listen. Der Anwender    |
| sollte primär auf die Funktion List (s.u.) zurückgreifen.                   |
+=============================================================================+*/
template <typename HEAD>
inline C<<HEAD,N> cons(const HEAD& h) {
  return C<HEAD,N>(h,N());
}

template <typename HEAD,typename TAIL>
inline C<HEAD,TAIL>  cons(const HEAD& h,const TAIL& t) {
  return C<HEAD,TAIL>( h, t);
}
```

```
template <typename HEAD,typename TAIL>
inline C<HEAD,TAIL>  cons_(const HEAD& h,const TAIL& t) {
  return C<HEAD,TAIL>( h, t);
}

/*
+=========================================================================+
| List                                                                    |
+=========================================================================+
| List ist eine mehrfach überladene Funktion zum Erstellen von Listen mit bis |
| zu 5 Elementen.                                                         |
+=========================================================================+*/
template <typename A1>
inline C<A1,N> List(const A1& a1) {
  return cons(a1);
}

template <typename A1,typename A2>
inline C<A1,C<A2,N> > List(const A1& a1,const A2& a2) {
  return cons(a1,cons(a2));
}

template <typename A1,typename A2,typename A3>
inline C<A1,C<A2,C<A3,N> > > List(const A1& a1,const A2& a2,const A3& a3) {
  return cons(a1, cons(a2,cons(a3)));
}

template <typename A1,typename A2,typename A3,typename A4>
inline C<A1,C<A2,C<A3,C<A4,N> > > > List(const A1& a1,const A2& a2,const A3& a3,const
A4& a4) {
  return cons(a1,cons(a2,cons(a3,cons(a4))));
}

template <typename A1,typename A2,typename A3,typename A4,typename A5>
inline C<A1,C<A2,C<A3,C<A4,C<A5,N> > > > > List(const A1& a1,const A2& a2,const A3&
a3,const A4& a4,const A5& a5) {
  return cons(a1, cons(a2,cons(a3,cons(a4,cons(a5)))));
}

/*
+=========================================================================+
| Compilezeit-Fehlermeldungen                                             |
+=========================================================================+
| Die folgenden Template-Deklarationen dienen dem Generieren von Fehler-  |
| meldungen.                                                              |
+=========================================================================+*/
template <typename LIST>
```

```
struct ERROR__FIRST_TEMPLATE_ARGUMENT_NEEDS_TO_BE_A_LIST_TYPE;

template <typename LIST>
struct ERROR__SECOND_TEMPLATE_ARGUMENT_NEEDS_TO_BE_A_LIST_TYPE;

/*
+=============================================================================+
| IsList                                                                      |
+=============================================================================+
| Prüft, ob es sich beim Argument um einen Listentyp (C oder N) handelt.      |
|                                                                             |
|    IsList a     = false                                                     |
|    IsList []    = true                                                      |
|    IsList (h:t) = true                                                      |
|    == Kann so natürlich nicht in Haskell realisiert werden!                 |
+=============================================================================+*/
template <typename T> struct
IsList  {  enum { Ret = false };};

template<> struct
IsList<N> { enum { Ret = true }; };

template <typename HEAD,typename TAIL> struct
IsList<C<HEAD,TAIL> > {enum { Ret = true };};

/*
+=============================================================================+
| Length                                                                      |
+=============================================================================+
| Berechnet die Länge einer Liste.                                            |
|                                                                             |
|   Length []    = 0                                                          |
|   Length (h:t) = 1 + Length t                                               |
+=============================================================================+*/
template <typename LIST>
struct Length {
  ERROR__FIRST_TEMPLATE_ARGUMENT_NEEDS_TO_BE_A_LIST_TYPE<LIST> error;
};

template <> struct
Length<N> { enum { Ret = 0 }; };

template <typename HEAD,typename TAIL> struct
Length<C<HEAD,TAIL> > {  enum { Ret = 1 + Length<TAIL>::Ret };};

template <typename LIST>
inline int length(const LIST&) {
```

```
  return Length<LIST>::Ret;
}

/*
+=============================================================================+
| At                                                                          |
+=============================================================================+
| Gibt das n'te Element einer Liste zurück.                                   |
|                                                                             |
|   At n []    = []                                                           |
|   At 0 (h:t) = h                                                            |
|   At n (h:t) = At (n-1) t                                                   |
+=============================================================================+*/
template <int n,typename LIST>
struct At {
  ERROR__SECOND_TEMPLATE_ARGUMENT_NEEDS_TO_BE_A_LIST_TYPE<LIST> error;
};

template <int n>
struct At<n,N> {
  typedef N RET;
};

template <typename HEAD,typename TAIL>
struct At<0,C<HEAD,TAIL> > {
  typedef HEAD RET;
};

template <int n,typename HEAD,typename TAIL>
struct At<n,C<HEAD,TAIL> > {
  typedef typename At<n-1,TAIL>::RET RET;
};

/*
+=============================================================================+
| Memeber                                                                     |
+=============================================================================+
| Prüft, ob eine Liste einen bestimmten Wert enthält.                         |
|                                                                             |
|    Member m []    = false                                                   |
|    Member m (h:t) = true,  if m==h                                          |
|                     false, otherwise                                        |
+=============================================================================+*/
template <typename M,typename LIST> struct
Member {
  ERROR__SECOND_TEMPLATE_ARGUMENT_NEEDS_TO_BE_A_LIST_TYPE<LIST> error;
};
```

```
template <typename M> struct
Member<M,N> {
  enum { Ret = false };
};

template <typename M,typename HEAD,typename TAIL> struct
Member<M,C<HEAD,TAIL> > {
  enum { Ret = Equal<M,HEAD>::Ret || Member<M,TAIL>::Ret };
};

/*
+=============================================================================+
| Drop                                                                        |
+=============================================================================+
| Entfernt die ersten 'n' Elemente einer Liste                                |
|                                                                             |
|   Drop n []     = []                                                        |
|   Drop n (h:t) = Drop (n-1) t, if (n>0)                                      |
|                = (h:t), otherwise                                           |
+=============================================================================+*/
template <int n,typename LIST>
struct Drop {
  ERROR__SECOND_TEMPLATE_ARGUMENT_NEEDS_TO_BE_A_LIST_TYPE<LIST> error;
};

template <int n> struct
Drop<n,N> {
  typedef N RET;
};

template <int n,typename HEAD,typename TAIL> struct
Drop<n,C<HEAD,TAIL> > {

  struct RemoveHead {
    template <typename Dummy>
    struct Apply {
      typedef typename Drop<n-1,TAIL>::RET RET;
    };
  };

  struct Finished {
    template <typename Dummy>
    struct Apply {
      typedef C<HEAD,TAIL> RET;
    };
  };
```

```
typedef typename If< (n>0),
                     RemoveHead,
                     Finished
                   >::RET::template Apply<int>::RET RET;

};

/*
+=============================================================================+
| Take                                                                        |
+=============================================================================+
| Extrahiert die ersten 'n' Elemente einer Liste.                             |
|                                                                             |
|   Take n []     = []                                                        |
|   Take n (h:t) = h:Take (n-1) t, if (n>0)                                    |
|                = [], otherwise                                              |
+=============================================================================+*/
template <int n,typename LIST>
struct Take {
  ERROR__SECOND_TEMPLATE_ARGUMENT_NEEDS_TO_BE_A_LIST_TYPE<LIST> error;
};

template <int n> struct
Take<n,N> {
  typedef N RET;
};

template <int n,typename HEAD,typename TAIL> struct
Take<n,C<HEAD,TAIL> > {
  typedef typename If< (n>0),
                     C<HEAD, typename Take<n-1,TAIL>::RET>,
                     N
                   >::RET RET;
};

#ifdef TMPL_LIST_NAMESPACE
}

#endif

#endif
```

B.3 Analyse von CoreEML-Programmen

```
/*
+=========================================================================+
| AppDepth                                                                |
+=========================================================================+
| Berechnet die Tiefe eines Applikationsknotens.                          |
|                                                                         |
|   AppDepth a = 0                                                        |
|   AppDepth App(F,A)     = 1 + AppDepeh F                                |
|   AppDepth App(Var N,A) = 0                                             |
+=========================================================================+*/
template <typename F> struct AppDepth;
template <int NAME,typename A>  struct
AppDepth<Var<NAME> >  { enum { Ret = 0 };};
template <typename F,typename A> struct
AppDepth<App<F,A> >   { enum { Ret = 1 + AppDepth<F>::Ret };};
/*
+=========================================================================+
| GetScName                                                               |
+=========================================================================+
| Ermittelt den Namen eines Superkombinators aus der linken Seite der zu= |
| gehörigen Superkombinatordefinition (App=Knoten).                       |
|                                                                         |
|   GetScName App(Var Name,A) = Var Name                                  |
|   GetScName App(F,A)        = GetScName F                               |
+=========================================================================+*/
// Ermitteln des innersten Funktionsnamens.
template <typename APP> struct GetScName;
template <int NAME,typename A> struct
GetScName<App<Var<NAME>,A> > { typedef Var<NAME> RET; };
template <typename F,typename A> struct
GetScName<App<F,A> > { typedef typename GetScName<F>::RET RET; };
/*
+=========================================================================+
| GetScDefinition                                                         |
+=========================================================================+
| Extrahiert die zu SCNAME gehörende Superkombinatordefinition aus dem    |
| CoreEML=Skript SKRIPT.                                                  |
|                                                                         |
|   GetScDefinition [] Name = error                                       |
|   GetScDefinition ((Sc LHS RHS):TAIL) Name = If (GetScName LHS) == Name |
|                                              then (Sc LHS RHS)          |
|                                              else GetScDefinition TAIL NAME |
+=========================================================================+*/
template <typename SCRIPT,int SCNAME> struct GetScDefinition;
```

```
template <int SCNAME> struct
GetScDefinition<TML::N, SCNAME> { typedef FAIL RET; };
template <typename LHS,typename RHS, typename TAIL int SCNAME> struct
GetScDefinition<TML::C< Sc<LHS,RHS>, TAIL>, SCNAME> {
  typedef typename If<Equal<typename getScName<LHS>::RET, Var<SCNAME> >::RET,
                      Sc<LHS,RHS>
                      typename GetScDefinition<TAIL, SCNAME>::RET
                     >::RET RET;  };
/*
+=============================================================================+
| GetArgPos                                                                   |
+=============================================================================+
| Berechnet, an welcher Stelle der Paramter Var<NAME> auf der linken Seite    |
| LHS einer Superkombinatordefinition auftaucht. Wird benötigt, um die Posi=  |
| tion eines Superkombinatorarguments auf dem Stack zu berechnen.             |
|                                                                             |
|   GetArgPos a NAME = 1                                                      |
|   GetArgPos App(F,(Var NAME)) NAME = 0                                      |
|   GetArgPos App(F,A) NAME = 1 + GetArgPos F NAME                            |
+=============================================================================+*/
template <typename LHS,int NAME> struct
GetArgPos { enum {Pos=1};};
template <typename F,int NAME> struct
GetArgPos<App<F,Var<NAME> >,Var<NAME> > { enum {Pos=0};};
template <typename F,typename A,int NAME> struct
GetArgPos<App<F,A>, NAME> { enum {Pos=1+GetArgPos<F,NAME>::Ret};};
```

B.4 Test auf Konvertierbarkeit

```
// Metaprogramm zum Test, ob Objekte vom Typ A in Objekte
// vom Typ B konvertiert werden können.
template <typename A,typename B> struct
chkConv {
  // Für die folgenden Strukturen gilt garantiert
  // sizeof(YES) != sizeof(NO)
  struct YES { char foo;                  };
  struct NO  { char foo[sizeof(int) * 2]; };
  // Kann B in A konvertiert werden, wird diese Methode selektiert ...
  static inline YES conv(B target);
  // ... diese Methode ist mit allen Typen kompatibel ...
  static inline NO conv( ... );

  // Methode zum "Anlegen" eines B-Objektes, ohne einen eventuell nicht
  // vorhandenen default-Konstruktor aufzurufen.
  static inline A Make_B_Object();

  // Trick: Argumente an sizeof werden nicht ausgewertet. Der
  //        Übersetzer muß nur den Argumenttyp kennen. Dieser
  //        hängt von der gewählten conv-Methode ab.
  enum { Ret = sizeof( conv( Make_B_Object() ) ) == sizeof(YES) };
};
```

Literaturverzeichnis

[1] *International Standard, Programming Languages - PASCAL, ISO/IEC 7185* . 1990

[2] *International Standard, Programming Languages - Fortran 95, ISO/IEC: 1539-1*. 1997

[3] *International Standard, Programming Languages - C++, ISO/IEC: 14882* . 1998

[4] *International Standard, Programming Languages - C, ISO/IEC 9899* . 1999

[5] Abadi, Martin ; Cardelli, Luka: *A Theory of Objects*. Springer Verlag, 1996

[6] Abelson, Harold ; Sussman, Gerald J. ; Sussman, Julie: *Struktur und Interpretation von Computerprogrammen*. Springer Verlag, 2001. – ISBN 35–404–2342–7

[7] Alexandrescu, Andrei: *Modern C++ Design*. Addison-Wesley, 2001. – ISBN 02–017043–15

[8] Ambler, Allen L. ; Burnett, Margaret M. ; Zimmermann, Betsy A.: Operational Versus Definitional: A Perspective on Programming Paradigms. In: *IEEE Computer* (1992), S. 28–43

[9] Augustsson, Lennart: *Compiling Lazy Functional Languages, Part II*, Chalmers University, Sweden, Diss., 1987

[10] Augustsson, Lennart: Implementing Haskell Overloading. In: *Functional Programming Languages and Computer Architecture*, 1993, S. 65–73

[11] Baader, Franz ; Nipkow, Tobias: *Term Rewriting and All That*. Cambridge University Press, 1998

[12] Barendregt, Henk: *The lambda calculus: its syntax and semantics*. North Holland, 1984

[13] Barendregt, Henk: Lambda Calculi with Types. In: Abramsky, S. (Hrsg.) ; Gabbay, D. M. (Hrsg.) ; T. S. E. Maibaum (Eds.), Clarendon (Hrsg.): *Handbook of Logic in Computer Science, Volume 2 (Background: Computational Structures)*. Oxford University Press, 1992, S. 117–309

[14] Baumgartner, Gerald ; Jansche, Martin ; Peisert, Christopher D.: Support for Functional Programming in Brew. In: Davis, Kei (Hrsg.) ; Smaragdakis, Yannis (Hrsg.) ; Striegnitz, Jörg (Hrsg.): *Multiparadigm Programming with Object-Oritented Languages* Bd. 7, 2001, S. 111–125

[15] Bjesse, Per ; Claessen, Koen ; Sheeran, Mary ; Singh, Satnam: Lava: Hardware Design in Haskell. In: *ACM SIGPLAN Notices* 34 (1999), Januar, Nr. 1, S. 174–184. – ISSN 0362–1340

[16] Bolton, Finton: *Pure Corba.* SAMS, 2001. – ISBN 06–72318–121

[17] Botorog, George H. ; Kuchen, Herbert: Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In: *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, IEEE Computer Society, 1996, S. 243–252

[18] Box, Dox ; Sells, Cris: *Essential .NET 1. The Common Language Runtime.* Addison-Wesley, 2002. – ISBN 02–017–3411–7

[19] Bracha, Gilad ; Odersky, Martin ; Stoutamire, David ; Wadler, Philip: Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In: Chambers, Craig (Hrsg.): *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA).* Vancouver, BC, 1998, S. 183–200

[20] Bruce, Kim B.: A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. In: *Journal of Functional Programming* 4 (1994), Nr. 2, S. 127–206

[21] Bruce, Kim B.: *Foundations of Object-Oriented Languages: Types and Semantics.* MIT Press, März 2002. – ISBN 0–26202–523–X

[22] Bruce, Kim B. ; Meyer, Albert ; Mitchell, John C.: The Semantics of the Second-Order Lambda Calculus. In: *Information and Computation* 85 (1990), März, Nr. 1, S. 76 ff.

[23] Bruce, Kim B. ; Schuett, Angela ; Gent, Robert van: PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. In: *Lecture Notes in Computer Science* 952 (1995), S. 27–51

[24] Budd, Timothy A.: *Multiparadigm Programming in Leda.* Addison Wesley, 1995. – ISBN 0–201–82080–3

[25] Budd, Timothy A.: The Return of Jensen's Device. In: Striegnitz, Jörg (Hrsg.) ; Davis, Kei (Hrsg.) ; Smaragdakis, Yannis (Hrsg.): *Multiparadigm Programming with Object-Oritented Languages* Bd. 13, 2002, S. 45–63

[26] Budd, Timothy A. ; Pandey, Rajeev K.: Never mind the Paradigm: What about Multiparadigm Languages? In: *SIGCSE Bulletin* 27 (1995), Nr. 2, S. 25–30

[27] Cann, D.C.: Retire FORTAN? A Debate Rekindled. In: *Communications of the ACM* 35 (1992), Nr. 8, S. 81–89

[28] Cardelli, Luca ; Martini, Simone ; Mitchell, John C. ; Scedrov, Andre: An extension of system F with subtyping. In: *Inf. Comput.* 109 (1994), Nr. 1-2, S. 4–56. – ISSN 0890–5401

[29] Cardelli, Luca ; Mitchell, John C.: Operations on Records. In: Gunter, C. A. (Hrsg.) ; Mitchell, J. C. (Hrsg.): *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* Cambridge, MA : The MIT Press, 1994, S. 295–350

[30] Cardelli, Luca ; Wegner, Peter: On Understanding Types, Data Abstraction, and Polymorphism. In: *ACM Computing Surveys* 17 (1985), Nr. 4, S. 471–522

[31] Cartwright, Robert ; Felleisen, Matthias: Extensible Denotational Language Specifications. In: Hagiya, Masami (Hrsg.) ; Mitchell, John C. (Hrsg.): *Theoretical Aspects of Computer Software* Bd. 789. Springer-Verlag, 1994, S. 244–272

[32] Castagna, Giuseppe ; Ghelli, Giorgio ; Longo, Giuseppe: A calculus for overloaded functions with subtyping. In: *Proceedings of the 1992 ACM conference on LISP and functional programming*, ACM Press, 1992. – ISBN 0–89791–481–3, S. 182–192

[33] Cleeland, Chris ; Schmidt, Douglas C. ; Harrison, Tim: External polymorphism. (1997), S. 377–390. ISBN 0–201–31011–2

[34] Code, Murray: *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989. – ISBN 0–262–53086–4

[35] Collins, George E.: A method for overlapping and erasure of lists. In: *Communications of the ACM* 12 (1960), S. 655–675

[36] Consel, C. ; Danvy, O.: Tutorial Notes on Partial Evaluation. In: *Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993* ACM, ACM Press, 1993, S. 493–501

[37] Conway, Melvin E.: Proposal for an UNCOL. In: *Communications of the ACM* 1 (1958), Oktober, Nr. 10, S. 5–8. – ISSN 0001–0782

[38] Courtney, Antony: Frappé: Functional Reactive Programming in Java. In: *Proceedings of Symposium on Practical Aspects of Declarative Languages* ACM, 2001

[39] Crary, Karl ; Weirich, Stephanie ; Morrisett, Greg: Intensional polymorphism in type-erasure semantics. In: *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ACM Press, 1998. – ISBN 1–58113–024–4, S. 301–312

[40] Czarnecki, Krzysztof ; O'Donnel, Jon ; Striegnitz, Jörg ; Taha, Walid: DSL Implementation in MetaOCaml, Template Haskell, and C++, Springer Verlag, 2004, S. 51–72

[41] Eifrig, Jonathan ; Smith, Scott F. ; Trifonov, Valery ; Zwarico, Amy E.: An Interpretation of Typed OOP in a Language with State. In: *Lisp and Symbolic Computation* 8 (1995), Nr. 4, S. 357–397

[42] Eisenscker, Ulrich W. ; Czarnecki, Krzysztof: *Generative Programming - Methods, Tools and Applications.* Addison Wesley, 2000

[43] Elliott, Conal ; Hudak, Paul: Functional Reactive Animation. In: *International Conference on Functional Programming*, 1997, S. 163–173

[44] Engel, Joshua: *Programming for the Java Virtual Machine*. Addison-Wesley, 1999. – ISBN 02–013–0972–6

[45] Finne, Sigbjorn ; Leijen, Daan ; Meijer, Erik ; Jones, Simon L. P.: H/Direct: A Binary Foreign Language Interface for Haskell. In: *International Conference on Functional Programming*, 1998, S. 153–162

[46] Finne, Sigbjorn ; Peyton Jones, Simon L.: Pictures: A Simple Structured Graphics Model. In: *Proceedings of Glasgow Functional Programming Workshop*, 1995

[47] Floyd, Robert W.: The Paradigms of Programming. In: *Communications of the ACM* 22 (1979), Nr. 8

[48] Foster, Ian (Hrsg.) ; Kesselman, Carl (Hrsg.): *The Grid : Blueprint For A New Computing Infrastructure*. Morgan Kaufmann, 1998

[49] Friedman, Daniel P. ; Felleisen, Matthias: *The Seasoned Schemer*. MIT Press, 1995

[50] Fukunaga, Koichi ; Hirose, Shin ichi: An experience with a Prolog-based object-oriented language. In: *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, 1986. – ISBN 0–89791–204–7, S. 224–231

[51] Futamura, Yoshihiko: Partial Evaluation of Computation: An approach to a compiler-compiler. In: *Systems, Computers, Controls* 2 (1971), Nr. 5, S. 45–50

[52] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison Wesley, 2001

[53] Garrigue, J. *Programming with polymorphic variants*. 1998

[54] Garrigue, Jacques: Code reuse through polymorphic variants. In: *Workshop on Foundations of Software Engineering. Sasaguri, Japan*, 2000

[55] Girard, Jean-Yves: *Interprétation functionelle et élimination des coupures de l'arithmétique d'ordre supérierur*, Universite Paris VII, Diss., 1972

[56] Glenstrup, Arne J. ; Jones, Neil D.: BTA Algorithms to Ensure Termination of Off-Line Partial Evaluation. In: *Proceedings of the Second International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Springer-Verlag, 1996. – ISBN 3–540–62064–8, S. 273–284

[57] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java Language Specification (2nd Edition)*. Addison Wesley, 2000

[58] Gropp, William ; Lusk, Erwing ; Skjellum, Anthony: *Using MPI - Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1988. – ISBN 0–262–57134–X

[59] Group, The O. *The COM / DCOM Reference*

[60] Guzman, Joel de ; Nuffer, Dan: The Spirit Parser Library: Inline Parsing in C++. In: *C/C++ Users Journal* 21 (2003), September, Nr. 9

[61] Hailpern, Brent: Multiparadigm Languages and Environments. In: *IEEE Software* 3 (1986), Nr. 1, S. 6 – 9

[62] Hall, Cordelia V. ; Hammond, Kevin ; Peyton Jones, Simon L. ; Wadler, Philip L.: Type classes in Haskell. In: *ACM Trans. Program. Lang. Syst.* 18 (1996), Nr. 2, S. 109–138. – ISSN 0164–0925

[63] Hallgren, T. *Fun with functional dependencies.* 2001

[64] Hanson, R. J. ; Krogh, F. T. ; Lawson, C. L.: Improving the Efficiency of Portable Software for Linear Algebra. In: *ACM SIGNUM Newsletter* 8 (1973), Nr. 4

[65] Hanus, Michael: A Unified Computation Model for Functional and Logic Programming. In: *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, 1997, S. 80–93

[66] Harper, Robert ; Mitchell, John C. ; Moggi, Eugenio: Higher-order modules and the phase distinction. In: *Conference record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages.* San Francisco, CA, January 1990, S. 341–354

[67] Harper, Robert ; Morrisett, Greg: Compiling Polymorphism Using Intensional Type Analysis. In: *22nd Symposium on Priciples of Programming Languages, POPL '95*, 1995, S. 130–141

[68] Harrison, W. ; Ossher, H. *Subject-Oriented Programming.* 1993

[69] Hayes, Roger: A Simple System for Constructing Distributed, Mixed-language Programs. In: *Software - Practice and Experience* 18 (1988), Nr. 7, S. 641–660

[70] Hinze, Ralf: *Einführung in die funktionale Programmierung mit Miranda.* B.G. Teubner Stuttgart, 1992

[71] Horspool, R. N. ; Levy, Michael R.: Translator-Based Multiparadigm Programming. In: *Journal of Systems and Software* 23 (1993), Nr. 1, S. 39–49

[72] Hudak, P. ; Makucevich, T. ; Gadde, S. ; Whong, B.: Haskore Music Notation – An Algebra of Music. In: *Journal of Functional Programming* 6 (1996), Mai, Nr. 3, S. 465–483

[73] Hudak, Paul: Building Domain Specific Embedded Languages. In: *ACM Computing Surveys* 28A (1996), Dezember, S. (electronic)

[74] Hudak, Paul ; Peterson, John ; Fasel, Joseph. *A Gentle Introduction To Haskell 98.* `http://www.haskell.org/tutorial/`. 1999

[75] Hughes, J.: Pretty-Printing: an Exercise in Functional Programming. In: Bird, R. S. (Hrsg.) ; Morgan, C. C. (Hrsg.) ; Woodcock, J. C. P. (Hrsg.): *Mathematics of Program Construction; Second International Conference; Proceedings.* Berlin, Germany : Springer-Verlag, 1993. – ISBN 3–540–56625–2, S. 11–13

[76] Hutton, G.: Combinator Parsing. In: *Journal of Functional Programming* (1993)

[77] Ihringer, Thomas: *Allgemeine Algebra*. B.G. Teubner, 1994. – ISBN 3–519–12083–6

[78] Jacobsen, Ivar ; Booch, Grady ; Rumbaugh, James: *The Unified Software Development Process*. Addison Wesley, 1999

[79] Järvi, Jaakko: C++ Function Object Binders Made Easy. In: *Proceedings of the Generative and Component-Based Software Engineering'99* Bd. 1799. Berlin, Germany : Springer, August 2000, S. 165–177

[80] Järvi, Jaakko ; Powell, Gary: The Lambda-Library: Lambda Abstractions in C++. In: *Second Workshop on C++ Template Programming, Erfurt, Germany*, 2001

[81] Jenkins, Michael A. ; Glasgow, Janice I. ; McCrosky, Carl D.: Programming Styles in Nial. In: *IEEE Software* 3 (1986), January, S. 46 – 55

[82] Johnsson, Thomas: *Compiling Lazy Functional Languages*, Chalmers University, Sweden, Diss., 1987

[83] Jones, Mark P.: Type classes with functional dependencies. In: Smolka, G. (Hrsg.): *Proceedings of the 9th European Symposium on Programming*, Springer, March 2000 (LNCS), S. 230–244

[84] Jones, Neil D.: Challenging Problems in Partial Evaluation and Mixed Computation. In: *Partial Evaluation and Mixed Computation*. North Holland, 1988, S. 1–14

[85] Jones, Neil D.: An Introduction to Partial Evaluation. In: *ACM Computing Surveys* 28 (1996), September, Nr. 3, S. 480–504

[86] Jones, Neil D.: What Not to Do When Writing an Interpreter for Specialisation. In: Danvy, Olivier (Hrsg.) ; Glück, Robert (Hrsg.) ; Thiemann, Peter (Hrsg.): *Partial Evaluation* Bd. 1110, Springer-Verlag, 1996, S. 216–237

[87] Jones, Neil D. ; Glenstrup, Arne J.: Program generation, termination, and binding-time analysis. In: *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, ACM Press, 2002. – ISBN 1–58113–487–8, S. 283–283

[88] Jones, Neil D. ; Gomard, Carsten K. ; Sestoft, Peter: *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. – ISBN 0–13–020249–5

[89] Jones, Richard ; Lins, Garbriel: *Garbage Collection*. John Wiely and Sons, 1996

[90] Kahl, Wolfram: Basic Pattern Matching Calculi / Sofware Quality Research Laboratory, Department of Computing and Software, McMaster University, Ontario, Canada. 2003. – Forschungsbericht

[91] Karmesin, Steve ; Crotinger, James ; Cummings, Julian ; Haney, Scott ; Humphrey, William J. ; Reynders, John ; Smith, Stephen ; Williams:, Timothy: Array Design and Expression Evaluation in POOMA II. In: Caromel, D. (Hrsg.) ; Oldehoeft, R. (Hrsg.) ; Tholburn, M. (Hrsg.): *Proceedings of ISCOPE'98* Bd. 1505, Springer Verlag, 1998

[92] Kiczales, Gregor ; Lamping, John ; Menhdhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: Akşit, Mehmet (Hrsg.) ; Matsuoka, Satoshi (Hrsg.): *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland* Bd. 1241. New York, NY : Springer-Verlag, 1997, S. 220–242

[93] Krishnamurthi, Shriram ; Felleisen, Matthias ; Friedman, Daniel P.: Synthesizing Object-Oriented and Functional Design to Promote Re-use, 1998, S. 91–113

[94] Kuchen, Herbert ; Striegnitz, Jörg: Higher-order functions and partial applications for a C++ skeleton library. In: *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, ACM Press, 2002. – ISBN 1–58113–599–8, S. 122–130

[95] Kuck, D. J. ; Kuhn, R. H. ; Leasure, B. ; Wolfe, M.: The structure of an advanced vectorizer for pipelined processors. In: *Proceedings of the COMPSAC 80, the 4th International Computer Software and Applications Conference*, 1980, S. 709–715

[96] Kuck, D. J. ; Kuhn, R. H. ; Padua, D. A. ; Leasure, B. ; Wolfe, M.: Dependence graphs and compiler optimizations. In: *Conference Record of the Eighth Annual ACM Symposium on the Principals of Programming Languages*, 1981, S. 207–218

[97] Kuhn, Thomas S.: *The Structure of Scientific Revolutions.* University of Chicago Press, 1962

[98] Landry, Walter: Implementing a High-Performance Tensor Library. In: Davis, Kei (Hrsg.) ; Striegnitz, Jörg (Hrsg.): *Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'01).* Tampa, Florida : Research Centre Jülich, Central Institute for Applied Mathematics, 2001

[99] Läufer, K.: A Framework for Higher-Order Functions in C++. In: *Proc. Conf. Object-Oriented Technologies (COOTS).* Monterey, CA : USENIX, June 1995

[100] Leijen, Daan ; Meijer, Erik: Domain specific embedded compilers. In: *Domain-Specific Languages*, 1999, S. 109–122

[101] Leijen, Daan ; Meijer, Erik: Domain specific embedded compilers. In: *Proceedings of the 2nd conference on Domain-specific languages*, ACM Press, 1999. – ISBN 1–58113–255–7, S. 109–122

[102] Levine, John R. ; Mason, Tony ; Brown, Doug: *Ley & yacc.* O'Reilly, 1992. – ISBN 1565920007

[103] Levy, Michael R.: Proposal for a Foreign Language Interface to Prolog. In: *Workshop on Logic Programming Environments*, 1995

[104] Liang, Sheng: *JAVA Native Interface: Programmer's Guide and Specification.* Addison-Wesley, 1999

[105] Lindholm, Tim ; Yellin, Frank: *The JAVA Virtual Machine Specification.* Addison-Wesley, 1999. – ISBN 02–014–3294–3

[106] Lippman, Stan: *Inside the C++ Object Model.* Addison Wesley, 1996

[107] Madsen, Ole L.: Towards a Unified Programming Language. In: Bertino, Elisa (Hrsg.): *ECOOP 2000 - Object-Oriented Programming* Bd. 1850, Springer Verlag, 2000, S. 1 – 26

[108] Makholm, Henning: On Jones-Optimal Specialization for Strongly Typed Languages. In: *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, Springer-Verlag, 2000. – ISBN 3–540–41054–6, S. 129–148

[109] Matthews, John ; Cook, Byron ; Launchbury, John: Microprocessor Specification in Hawk. In: *Proceedings of the 1998 International Conference on Computer Languages*, IEEE Computer Society Press, 1998. – ISBN 0–780–35005–7, 0–8186–8454–2, 0–8186–8456–9, S. 90–101

[110] Mauny, M.: Parsers and Printers as Stream Destructors Embedded in Functional Languages. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* ACM/IFIP, 1989, S. 360–370

[111] McNamara, Brian ; Smaragdakis, Yannis: Functional programming in C++. In: *ACM SIGPLAN Notices* 35 (2000), Nr. 9, S. 118–129

[112] *MetaOCaml: A Compiled, Type-safe Multi-stage Programming Language*. Available online from `http://www.metaocaml.org/`. 2003

[113] Meyer, Bertrand: *Eiffel: the language*. Prentice-Hall, 1991. – ISBN 0–13247–925–7

[114] Mogenson, Torben: Glossary for Partial Evaluation and Related Topics. In: *Higher-Order and Symbolic Computation* 13 (2000), Nr. 4

[115] Moggi, Eugenio: Computational Lambda-Calculus and Monads. In: *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*. Washington, DC : IEEE Computer Society Press, 1989, S. 14–23

[116] Mohr, Bernd ; Striegnitz, Jörg: Objektorientiertes Programmieren in C++ / Zentrum für Hochleistungsrechnen, TU Dresden. 2000 (ZHR-IR-2002). – Forschungsbericht

[117] Morrisett, Greg ; Tarditi, David ; Cheng, Perry ; Stone, Chris ; Harper, Robert ; Lee, Peter. *The TIL/ML Compiler: Performance and Safety through Types*. 1996

[118] Morrisett, Greg: *Compiling with Types*, Carnegie Mellon University, Diss., 1995

[119] Muchnik, Steve: *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997

[120] Müller, Martin ; Müller, Tobias ; Van Roy, Peter: Multi-ParadigmProgramming in Oz. In: Smith, Donald (Hrsg.) ; Ridoux, Olivier (Hrsg.) ; Van Roy, Peter (Hrsg.): *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog*. Portland, Oregon, 7 1995. – A Workshop in Association with ILPS'95

[121] Nelson, Philip A.: A comparison of PASCAL intermediate languages. In: *Proceedings of the SIGPLAN symposium on Compiler construction*, 1979. – ISBN 0–89791–002–8, S. 208–213

[122] Neubauer, Matthias ; Thiemann, Peter ; Gasbichler, Martin ; Sperber, Michael: Functional logic overloading. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, 2002. – ISBN 1–58113–450–9, S. 233–244

[123] Nielson, Flemming ; Nielson, Hanne R.: *Two-level functional languages*. Cambridge University Press, 1992. – ISBN 0–521–40384–7

[124] Nielson, Hanne R. ; Nielson, Flemming: *Semantics with Applications: A Formal Introduction*. Chichester : John Wiley & Sons, 1992 (Wiley Series in Data Communications and Networking for Computer Progammers)

[125] Nielson, Henning ; Riis Nielson, Hanne ; Hankin, Chris: *Principles of Program Analysis*. Springer-Verlag, 1999. – ISBN 3–540–65410–0

[126] Nordlander, Johan ; Carlsson, Magnus: Reactive Objects in a Functional Language – An escape from the evil „I". In: *Proceedings of the Haskell Workshop, Amsterdam, Holland*, 1997

[127] Odersky, Martin ; Runne, Enno ; Wadler, Philip: Two Ways to Bake Your Pizza - Translating Parameterised Types into JAVA. In: Jazayeri, M. (Hrsg.) ; Loos, R. (Hrsg.) ; Musser, D. (Hrsg.): *Generic Programming '96*. Springer-Verlag, 2000 (LNCS 1766), S. 114–132

[128] Odersky, Martin ; Wadler, Philip: Pizza into Java: Translating Theory into Practice. In: *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, ACM Press, New York (NY), USA, 1997, S. 146–159

[129] O'Donnell, John: Overview of Hydra: A Concurrent Language for Synchronous Digital Circuit Design. In: *Proceedings 16th International Parallel & Distributed Processing Symposium*, IEEE Computer Society, April 2002. – Workshop on Parallel and Distribued Scientific and Engineering Computing with Applications—PDSECA. – ISBN 0–7695–1573–8, S. 234 (abstract)

[130] Okasaki, Chris: Even higher-order functions for parsing or Why would anyone ever want to use a sixth-order function? In: *Journal of Functional Programming* 8 (1998), März, Nr. 2, S. 195–199

[131] *Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA.* Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`

[132] Paulson, Larry C.: *ML for the Working Programmer*. Cambridge University Press, 1986

[133] Perry, Nigel ; Meijer, Erik. *Implementing Funtional Languages on Object-Oriented Virtual Machines (draft).* Available online from `http://research.microsoft.com/ emeijer`

[134] Peterson, J. ; Hager, G. ; Hudak, P.: A Language for Declarative Robotic Programming. In: *Proceedings of IEEE Conf. on Robotics and Automation*, 1999

[135] Peyton Jones, Simon: Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. In: *Journal of Functional Programming* 2 (1992), Nr. 2, S. 127–22

[136] Peyton Jones, Simon ; Hughes, John. *Report on the Programming Language Haskell 98*. 1999

[137] Peyton Jones, Simon ; Jones, Mark ; Meijer, Erik: Type Classes: An exploration of the design space. In: Launchbury, J. (Hrsg.): *Proceedings of the Haskell Workshop*, 1997

[138] Peyton Jones, Simon ; Nordin, Thomas ; Reid, Alastair: Greencard: a foreign language interface Haskell. In: *Haskell Workshop*. Amsterdam, The Netherlands : ACM, June 1997

[139] Peyton Jones, Simon L.: *The Implementation of Functional Programming Languages*. Prentice Hall, April 1987

[140] Peyton Jones, Simon L. ; Lester, David R.: *Implementing Funtional Languages: A Tutorial*. Prentice Hall, Ausgust 1992

[141] Peyton Jones, Simon L. ; Wadler, Philip: Imperative Functional Programming. In: *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, 1993, S. 71–84

[142] Pierce, Benjamin C.: *Types and Programming Languages*. MIT Press, 2002. – ISBN 02–621–6209–1

[143] Placer, John: Multiparadigm Research: A New Direction in Language Design. In: *SIGPLAN Notices* 16 (1991), March, Nr. 3, S. 9–17

[144] Rechenberg, Peter: Programming Languages as Thought Models. In: *Structured Programming* 11 (1990), S. 105–115

[145] Remy, Didier ; Vouillon, Jerome: Objective ML: An Effective Object-Oriented Extension to ML. In: *Theory and Practice of Object Systems* 4 (1998), Nr. 1, S. 27–50

[146] Reynolds, John C.: Towards a Theory of Type Structure. In: Robinet, B. (Hrsg.): *Programming Symposium* Bd. 19. Berlin : Springer-Verlag, 1974, S. 408–425

[147] Reynolds, John C.: Introduction to Part II, Polymorphic Lambda Calculus. In: Huet, Gérard (Hrsg.): *Logical Foundations of Functional Programming*. Reading, Massachusetts : Addison-Wesley, 1990, S. 77–86

[148] Schöning, Uwe: *Logik für Informatiker*. Spektrum Akademischer Verlag, 1995

[149] Seefried, Sean ; Chakravarty, Manuel ; Keller, Gabriele: *Optimising Embedded DSLs using Template Haskell*. – 15th International Workshop on the Implementation of Functional Languages, Edinburgh, Schottland, 2003

[150] Shao, Zhong: An Overview of the FLINT/ML Compiler. In: *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*. Amsterdam, The Netherlands, 1997

[151] Sheard, Tim ; Peyton Jones, Simon: Template Meta-programming for Haskell. In: Chakravarty, Manuel M. (Hrsg.): *ACM SIGPLAN Haskell Workshop 02*, ACM Press, Oktober 2002, S. 1–16

[152] Siek, Jeremy G. ; Lumsdaine, Andrew: The Matrix Template Library: Generic Components for High-performance Scientific Computing. In: *Computing in Science and Engineering* 1 (1999), Nov/Dec, Nr. 6, S. 70–78

[153] Steel, T.B.: Universal computer-oriented language. In: *Communications of the ACM* 4 (1961), March, Nr. 3, S. 138 ff. – ISSN 0001–0782

[154] Sterling, L. ; Shapiro, E.: *The Art of Prolog.* MIT Press, 1987

[155] Striegnitz, Jörg: Higher-Order Functions and Partial Application in C++ / Forschungszentrum Jülich, Zentralinstitut für angewandte Mathematik. 1999 (IB-9913). – Forschungsbericht

[156] Striegnitz, Jörg: Writing efficient programs with C++ / Zentrum für Hochleistungsrechnen, TU Dresden. 2000 (ZHR-IR-2010). – Forschungsbericht

[157] Striegnitz, Jörg ; Smith, Stephen A.: An Expression Template aware Lambda Function. In: *First Workshop on C++ Template Programming, Erfurt, Germany*, 2000

[158] Stroustrup, B.: *The C++ Programming Language.* Addison Wesley, 1997. – ISBN 0–201–88954–4

[159] Taha, Walid: *Multi-Stage Programming: Its Theory and Applications*, Oregon Graduate Institute of Science and Technology, Diss., 1999. – Available from [131]

[160] Taha, Walid ; Makholm, Henning ; Hughes, John: Tag Elimination and Jones-Optimality. In: *Lecture Notes in Computer Science* 2053 (2001), S. 257+

[161] Taha, Walid ; Sheard, Tim: Multi-Stage Programming with Explicit Annotations. In: *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM).* Amsterdam : ACM Press, 1997, S. 203–217

[162] Tarditi, D. ; Morrisett, G. ; Cheng, P. ; Stone, C. ; Harper, R. ; Lee, P.: TIL: A Type-Directed Optimizing Compiler for ML. In: *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, 1996, S. 181–192

[163] Thiemann, Peter: Programmable Type Systems for Domain Specific Languages. In: Comini, Marco (Hrsg.) ; Falaschi, Moreno (Hrsg.): *Electronic Notes in Theoretical Computer Science* Bd. 76, Elsevier, 2002

[164] Thiemann, Peter: A typed representation for HTML and XML documents in Haskell. In: *Journal of Functional Programming* 12 (2002), July, Nr. 4 and 5, S. 435–468

[165] Thompson, Simon: *Haskell: The Craft of Functional Programming.* Addison Wesley, 1999

[166] Unruh, Erwin: 1994. – Prime number computation. ANSI X3J16-94-0075/SO WG21-462

[167] Unruh, Erwin. *Template Metaprogrammierung.* Available online from `http://www.erwin-unruh.de/meta.html`. 2002

[168] Vandervoorde, David ; Josuttis, Nicolai M.: *C++ Templates.* Addison Wesley, 2002. – ISBN 02–017–3484–2

[169] Veldhuizen, Todd L.: Expression templates. In: *C++ Report* 7 (1995), Nr. 5, S. 26–31

[170] Veldhuizen, Todd L.: Template Metaprograms. In: *C++ Report* 7 (1995), Nr. 4, S. 36–43

[171] Veldhuizen, Todd L.: Arrays in Blitz++. In: *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98).* Berlin, Heidelberg, New York, Tokyo : Springer-Verlag, 1998

[172] Veldhuizen, Todd L. ; Gannon, Dennis: Active Libraries: Rethinking the roles of compilers and libraries. In: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98).* Philadelphia, PA, USA : SIAM, 1998

[173] Wadler, Philip: Monads for functional programming. In: Broy, M. (Hrsg.): *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*, Springer-Verlag, 1993

[174] Wadler, Philip ; Taha, Walid ; McQueen, David: How to add laziness to a strict language, without even being odd. In: *Workshop on Standard ML.* Baltimore, September 1998

[175] Wadsworth, Chris P.: *Semantics and Pragmatics of the Lambda Calculus*, Oxford University, Diss., 1971

[176] Warren, David H.: An Abstract Prolog Instruction Set / SRI International. 1983 (309). – Technical Note

[177] Weirich, Stephanie: *Programming With Types*, Cornell University, Diss., August 2002

[178] Widera, Manfred: *Complete Type Inference in Functional Programming.* Mensch & Buch Verlag, Berlin, 2001. – (PhD thesis, Dept. of Computer Science, FernUniversität Hagen)

[179] Winskel, Glynn: *Formal Semantics.* MIT Press, 1993

[180] Winston, Patrick H.: *On to Smalltalk.* Addison Wesley, 1997

[181] Zave, Pamela: A Compisitional Approach to Multiparadigm Programming. In: *IEEE Software* 9 (1989), November, S. 15–25

[182] Zenger, Matthias ; Odersky, Martin: Implementing Extensible Compilers. In: Davis, Kei (Hrsg.) ; Striegnitz, Jörg (Hrsg.): *Multiparadigm Programming with Object-Oriented Languages – MPOOL'01* Bd. 7, John von Neumann Institute for Computing, 2001

Lebenslauf

8.8.1970	Geboren in Thuine (bei Lingen/Ems)
1977 - 1980	Katholische Grundschule Wuppertal-Rondsdorf
1980 - 1981	Grundschule Marper Schulweg Wuppertal-Barmen
1981 - 1989	Städt. Gymnasium Erftstadt-Lechenich
1989 - 1991	Tagesheimgymnasium Kerpen (Abschluss: Allgemeine Hochschulreife)
1993 - 1999	Studium der Informatik an der Rheinisch-Westfälischen Technischen Hochschule Aachen (Abschluss: Diplom-Informatiker)
seit 1999	Wissenschaftlicher Mitarbeiter am Zentralinstitut für Angewandte Mathematik Forschungszentrum Jülich GmbH
2000	Dreimonatiger Forschungsaufenthalt am Advanced Computing Laboratory Los Alamos National Laboratory, Los Alamos, USA

Bisher sind erschienen:

Modern Methods and Algorithms of Quantum Chemistry - Proceedings
Johannes Grotendorst (Hrsg.)
Winterschule, 21. - 25. Februar 2000, Forschungszentrum Jülich
NIC-Serie Band 1
ISBN 3-00-005618-1, Februar 2000, 562 Seiten
nicht mehr lieferbar

Modern Methods and Algorithms of Quantum Chemistry - Poster Presentations
Johannes Grotendorst (Hrsg.)
Winterschule, 21. - 25. Februar 2000, Forschungszentrum Jülich
NIC-Serie Band 2
ISBN 3-00-005746-3, Februar 2000, 77 Seiten
nicht mehr lieferbar

Modern Methods and Algorithms of Quantum Chemistry - Proceedings, Second Edition
Johannes Grotendorst (Hrsg.)
Winterschule, 21. - 25. Februar 2000, Forschungszentrum Jülich
NIC-Serie Band 3
ISBN 3-00-005834-6, Dezember 2000, 638 Seiten

Nichtlineare Analyse raum-zeitlicher Aspekte der hirnelektrischen Aktivität von Epilepsiepatienten
Jochen Arnold
NIC-Serie Band 4
ISBN 3-00-006221-1, September 2000, 120 Seiten

Elektron-Elektron-Wechselwirkung in Halbleitern: Von hochkorrelierten kohärenten Anfangszuständen zu inkohärentem Transport
Reinhold Lövenich
NIC-Serie Band 5
ISBN 3-00-006329-3, August 2000, 146 Seiten

Erkennung von Nichtlinearitäten und wechselseitigen Abhängigkeiten in Zeitreihen
Andreas Schmitz
NIC-Serie Band 6
ISBN 3-00-007871-1, Mai 2001, 142 Seiten

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Kei Davis, Yannis Smaragdakis, Jörg Striegnitz (Hrsg.)
Workshop MPOOL, 18. Mai 2001, Budapest
NIC-Serie Band 7
ISBN 3-00-007968-8, Juni 2001, 160 Seiten

Europhysics Conference on Computational Physics - Book of Abstracts
Friedel Hossfeld, Kurt Binder (Hrsg.)
Konferenz, 5. - 8. September 2001, Aachen
NIC-Serie Band 8
ISBN 3-00-008236-0, September 2001, 500 Seiten

NIC Symposium 2001 - Proceedings
Horst Rollnik, Dietrich Wolf (Hrsg.)
Symposium, 5. - 6. Dezember 2001, Forschungszentrum Jülich
NIC-Serie Band 9
ISBN 3-00-009055-X, Mai 2002, 514 Seiten

Quantum Simulations of Complex Many-Body Systems: >From Theory to Algorithms - Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Hrsg.)
Winterschule, 25. Februar - 1. März 2002, Rolduc Conference Centre, Kerkrade, Niederlande
NIC-Serie Band 10
ISBN 3-00-009057-6, Februar 2002, 548 Seiten

Quantum Simulations of Complex Many-Body Systems: >From Theory to Algorithms - Poster Presentations
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Hrsg.)
Winterschule, 25. Februar - 1. März 2002, Rolduc Conference Centre, Kerkrade, Niederlande
NIC-Serie Band 11
ISBN 3-00-009058-4, Februar 2002, 194 Seiten

Strongly Disordered Quantum Spin Systems in Low Dimensions: Numerical Study of Spin Chains, Spin Ladders and Two-Dimensional Systems
Yu-cheng Lin
NIC-Serie Band 12
ISBN 3-00-009056-8, Mai 2002, 146 Seiten

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Jörg Striegnitz, Kei Davis, Yannis Smaragdakis (Hrsg.)
Workshop MPOOL 2002, 11. Juni 2002, Malaga
NIC-Serie Band 13
ISBN 3-00-009099-1, Juni 2002, 132 Seiten

Quantum Simulations of Complex Many-Body Systems: >From Theory to Algorithms - Audio-Visual Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Hrsg.)
Winterschule, 25. Februar - 1. März 2002, Rolduc Conference Centre, Kerkrade, Niederlande
NIC-Serie Band 14
ISBN 3-00-010000-8, November 2002, DVD

Numerical Methods for Limit and Shakedown Analysis
Manfred Staat, Michael Heitzer (Hrsg.)
NIC-Serie Band 15
ISBN 3-00-010001-6, Februar 2003, 306 Seiten

Design and Evaluation of a Bandwidth Broker that Provides Network Quality of Service for Grid Applications
Volker Sander
NIC-Serie Band 16
ISBN 3-00-010002-4, Februar 2003, 208 Seiten

Automatic Performance Analysis on Parallel Computers with SMP Nodes
Felix Wolf
NIC-Serie Band 17
ISBN 3-00-010003-2, Februar 2003, 168 Seiten

Haptisches Rendern zum Einpassen von hochaufgelösten Molekülstrukturdaten in niedrigaufgelöste Elektronenmikroskopie-Dichteverteilungen
Stefan Birmanns
NIC-Serie Band 18
ISBN 3-00-010004-0, September 2003, 178 Seiten

Auswirkungen der Virtualisierung auf den IT-Betrieb
Wolfgang Gürich (Hrsg.)
GI Conference, 4. - 5. November 2003, Forschungszentrum Jülich
NIC-Serie Band 19
ISBN 3-00-009100-9, Oktober 2003, 126 Seiten

NIC Symposium 2004
Dietrich Wolf, Gernot Münster, Manfred Kremer (Hrsg.)
Symposium, 17. - 18. Februar 2004, Forschungszentrum Jülich
NIC-Serie Band 20
ISBN 3-00-012372-5, Februar 2004, 482 Seiten

Measuring Synchronization in Model Systems and Electroencephalographic Time Series from Epilepsy Patients
Thomas Kreutz
NIC-Serie Band 21
ISBN 3-00-012373-3, Februar 2004, 138 Seiten

Computational Soft Matter: From Synthetic Polymers to Proteins - Poster Abstracts
Norbert Attig, Kurt Binder, Helmut Grubmüller, Kurt Kremer (Hrsg.)
Winterschule, 29. Februar - 6. März 2004, Gustav-Stresemann-Institut Bonn
NIC-Serie Band 22
ISBN 3-00-012374-1, Februar 2004, 120 Seiten

Computational Soft Matter: From Synthetic Polymers to Proteins - Lecture Notes
Norbert Attig, Kurt Binder, Helmut Grubmüller, Kurt Kremer (Hrsg.)
Winterschule, 29. Februar - 6. März 2004, Gustav-Stresemann-Institut Bonn
NIC-Serie Band 23
ISBN 3-00-012641-4, Februar 2004, 440 Seiten

Synchronization and Interdependence Measures and their Applications to the Electroencephalogram of Epilepsy Patients and Clustering of Data
Alexander Kraskov
NIC-Serie Band 24
ISBN 3-00-013619-3, Mai 2004, 106 Seiten

High Performance Computing in Chemistry
Johannes Grotendorst (Hrsg.)
Bericht des Verbundprojekts:
High Performance Computing in Chemistry - HPC-Chem
NIC-Serie Band 25
ISBN 3-00-013618-5, Dezember 2004, 160 Seiten

Alle Bände stehen online zur Verfügung unter http://www.fz-juelich.de/nic-series/.